**WEEK-2**

**ALGORITHMS AND DATA STRUCTURES**

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

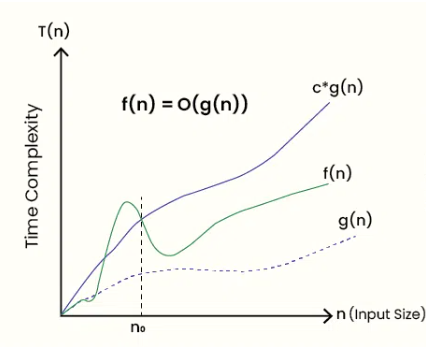
1a).Explain Big O notation and how it helps in analyzing algorithms.

**Big O notation** is used to describe the upper bound of an algorithm’s time or space complexity in terms of input size. It expresses the worst-case scenario for how an algorithm performs as the input size,n, grows large.

It’s denoted as **O(f(n))**, where f(n) describes how the time or space scales with input size.  
 Mathematically,  
   **f(n) ≤ c × g(n)** for all **n ≥ n₀**, where **c > 0**, and **g(n)** is the bounding function.

This allows developers to:

* Compare algorithm performance.
* Predict scalability.
* Choose the most efficient algorithm for the context.



1b).Describe the best, average, and worst-case scenarios for search operations.

Linear Search on an unsorted array of size n:

* **Best-case:**

The element is found at the **first position**.

**Time: O(1)**

* **Average-case:** The element is somewhere **in the middle (or randomly positioned)**.  
   On average, **n/2 comparisons** → **Time: O(n)**
* **Worst-case:** The element is **not present** or is at the **last position**.  
   Requires **n comparisons** → **Time: O(n)**

### Binary Search on a sorted array of size n

* **Best-case:** The **middle** **element** is the **target.  
   Time: O(1)**
* **Average-case:**Element found after splitting the array about **log₂(n) times**

**Time: O(log n)**

* **Worst-case:**Element **not present** or found **after all possible splits**.

**Time: O(log n)**

**2,3.Code:**

**Product.java**

package project;

public class Product {

int productId;

String productName;

String productCategory;

Product(int productId, String productName, String productCategory) {

this.productId = productId;

this.productName = productName;

this.productCategory = productCategory;

}

@Override

public String toString() {

return "ID: " + productId + ", Name: " + productName + ", Category: " + productCategory;

}

}

**MainClass.java**

package project;

import java.util.\*;

public class MainClass {

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

for (Product product : products) {

if (product.productId == key) return product;

}

return null;

}

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (products[mid].productId == key) return products[mid];

else if (products[mid].productId < key) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Product[] products = {

new Product(103, "HeadPhones", "Electronics"),

new Product(101, "Earrings", "Jewelry"),

new Product(105, "Pot", "Utensils"),

new Product(102, "Eraser", "Stationery"),

new Product(104, "Book", "Stationery")

};

System.out.print("Enter Product ID to search: ");

int idToSearch = scanner.nextInt();

Product found = linearSearch(products, idToSearch);

System.out.println(found != null ? "Linear Search: " + found : "Linear Search: Product with ID " + idToSearch + " not found");

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

Product foundB = binarySearch(products, idToSearch);

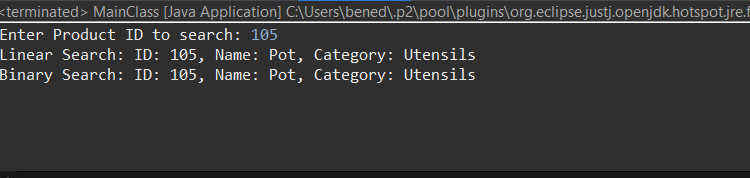
System.out.println(foundB != null ? "Binary Search: " + foundB : "Binary Search: Product with ID " + idToSearch + " not found");

}

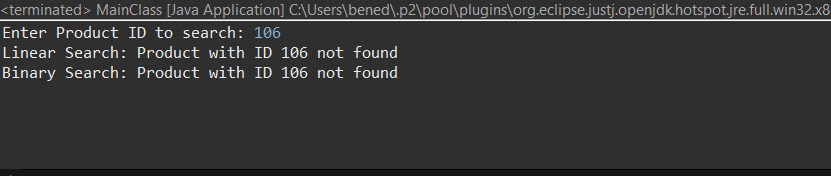
}

**Output:**

**Element found case:**

****

**Element not found case:**

****

4a).Compare the time complexity of linear and binary search algorithms.

The time complexity of linear search is **O(n)**, as it checks each element one by one. Binary search has a time complexity of **O(log n)** because it divides the search space in half each time. Binary search is significantly faster for large sorted datasets, while linear search works on both sorted and unsorted data.

4b).Discuss which algorithm is more suitable for your platform and why.

Since my platform uses a fixed array of products and searches by productId, **binary search** is a better choice. I can sort the array once and then search quickly in **O(log n)** time, which is much faster as the number of products grows.

**Exercise 7: Financial Forecasting**

1.Explain the concept of recursion and how it can simplify certain problems.

**Recursion** is a programming technique where a function calls itself to solve smaller parts of a larger problem. It simplifies problems that have a repetitive, self-similar structure — like computing factorials, Fibonacci numbers, or traversing trees. By breaking a problem into smaller versions of itself, recursion makes the solution easier to write and understand. The process continues until a **base case** is reached, which stops further calls. This method is especially useful when a problem can be naturally expressed in terms of smaller subproblems.

**2,3.Code:**

**FinancialForecast.java**

package project1;

public class FinancialForecast {

public static double forecast(double currentValue, double rate, int years) {

if (years == 0) return currentValue;

return forecast(currentValue, rate, years - 1) \* (1 + rate);

}

}

**Main.java**

package project1;

import java.util.\*;

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter current amount (₹): ");

double currentAmount = sc.nextDouble();

System.out.print("Enter annual growth rate (e.g., 0.08 for 8%): ");

double growthRate = sc.nextDouble();

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

int years = sc.nextInt();

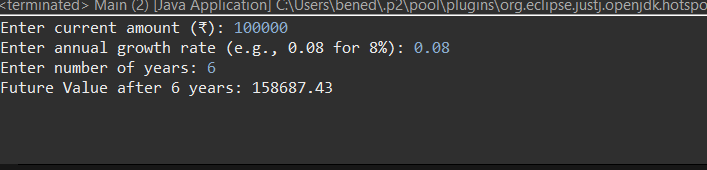
double future = FinancialForecast.forecast(currentAmount, growthRate, years);

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

}

}

**Output:**

****

4 a).Discuss the time complexity of your recursive algorithm.

The time complexity of my recursive solution is **O(n)** since it makes **n** **recursive calls** for n years.For each call, it computes the value of the previous year and then multiplies it by **(1 + rate)**, so the number of steps increases **linearly** with the number of years. There are no overlapping subproblems — simply one call per year.

4b).Explain how to optimize the recursive solution to avoid excessive computation.

To avoid excessive computation and the risk of a **StackOverflowError** ,the recursive solution can be optimized by replacing it with an **iterative (loop-based) approach**.Since the future value formula does not involve overlapping subproblems, there is **no need for recursion or memoization**. Each year simply multiplies the previous value by (1+rate), which can be easily handled using a for loop.

Iterative approach

public static double forecast(double currentValue, double rate, int years) {

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

currentValue \*= (1 + rate);

}

return currentValue;}