**Week 1 - Algorithms and Data Structures SupersetId-6431499**

**Name : Balaji V**

**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.

**1. Understand Asymptotic Notation:**

Big O notation is a mathematical representation used in computer science to describe the performance or complexity of an algorithm. It expresses how the runtime or space requirements grow as the size of the input (n) increases.

Big O helps us:

* Compare algorithms based on efficiency.
* Predict how algorithms will scale with larger inputs.
* Choose the best approach for solving a problem, especially when performance is critical.

### **1.Linear Search (Unsorted Array)**

Linear search checks each element one-by-one from start to end.

#### **Best Case:**

* **Scenario**: The element is at thefirst position.
* **Time Complexity**: O(1) (constant time)

#### **Average Case:**

* **Scenario**: The element is somewhere in the middle or distributed randomly.
* **Time Complexity**: O(n) (linear time)

#### **Worst Case:**

* **Scenario**: The element is at the last position or not present at all.
* **Time Complexity**: O(n)

### **2.Binary Search (Sorted Array Only)**

Binary search divides the array into halves to eliminate large sections each step.

#### **Best Case:**

* **Scenario**: The element is at the middle of the array.
* **Time Complexity**: O(1)

#### **Average Case:**

* **Scenario**: The element is located somewhere randomly in the array.
* **Time Complexity**: O(log n)

#### **Worst Case:**

* **Scenario**: The element is not in the array, or found at the very last level of search.
* **Time Complexity**: O(log n)

**2.Code:**

import java.util.\*;

public class ECommercePlatform{

static class Product implements Comparable<Product>{

private final int productId;

private final String productName;

private final 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 int compareTo(Product other){

return Integer.compare(this.productId, other.productId);

}

@Override

public String toString(){

return String.format("ProductID: %d, Name: %s, Category: %s", productId, productName, category);

}

}

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

for(Product product : products){

if(product.getProductId()==targetId){

return product;

}

}return null;

}

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].getProductId() == targetId){

return products[mid];

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

left=mid+1;

}else{

right=mid-1;

}

}return null;

}

public static void displayProducts(Product[] products){

System.out.println("\nAvailable Products:");

for(Product p : products){

System.out.println(p);

}

}

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

Product[] products = {

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

new Product(101, "Laptop", "Computers"),

new Product(103, "Sneakers", "Fashion"),

new Product(104, "Wrist Watch", "Accessories"),

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

};

displayProducts(products);

System.out.print("\nEnter the Product ID to search: ");

int targetId=sc.nextInt();

long startLinear=System.nanoTime();

Product resultLinear=linearSearch(products, targetId);

long endLinear=System.nanoTime();

Product[] sortedProducts=Arrays.copyOf(products, products.length);

Arrays.sort(sortedProducts);

long startBinary=System.nanoTime();

Product resultBinary=binarySearch(sortedProducts, targetId);

long endBinary=System.nanoTime();

System.out.println("\nSearch Results");

System.out.println("Linear Search Result : " + (resultLinear != null ? resultLinear : "Product not found"));

System.out.println("Binary Search Result : " + (resultBinary != null ? resultBinary : "Product not found"));

System.out.println("\nTime Analysis (in nanoseconds)");

System.out.println("Linear Search Time : " + (endLinear-startLinear));

System.out.println("Binary Search Time : " + (endBinary-startBinary));

System.out.println("\nAlgorithm Analysis");

System.out.println("Linear Search: Time Complexity -> O(n)");

System.out.println("Binary Search: Time Complexity -> O(log n)");

System.out.println("\n6431499--Balaji V--CTS");

System.out.println("\nRecommendation");

System.out.println("For small or unsorted datasets, Linear Search is acceptable.");

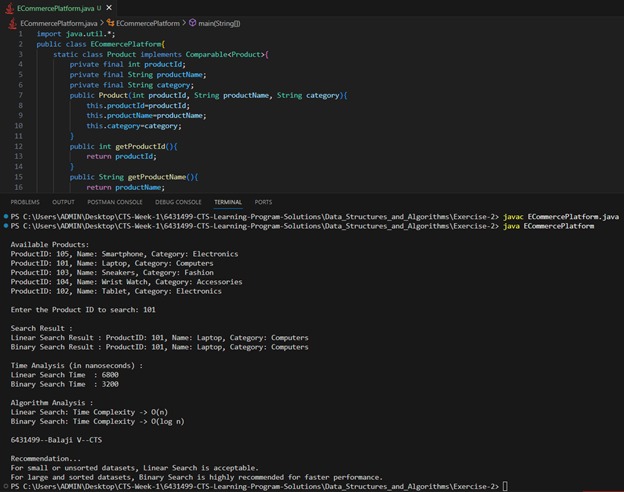
System.out.println("For large and sorted datasets, Binary Search is highly recommended for faster performance.");

sc.close();

}

}

**Output:**



**Exercise 7:** Financial Forecasting

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data**.**

**1.Understand Recursive Algorithms:**

**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 instances of a problem. It continues calling itself until it reaches a base case, which stops further recursion.

**Key Components:**

1. Base Case:  
   The condition under which the function stops calling itself.
2. Recursive Case:  
   The part where the function calls itself with a smaller or simpler input.

Recursion simplifies problems that can be broken down into smaller, similar subproblems. It provides a clear and logical approach for solving problems like:

* Mathematical calculations (e.g., factorial, Fibonacci numbers)
* Data structure traversal (e.g., trees, graphs)
* Divide and conquer algorithms (e.g., Merge Sort, Quick Sort)
* Backtracking problems (e.g., solving mazes, N-Queens)

**2.Code**

import java.util.\*;

public class FinancialForecasting{

public static double calculateFutureValueRecursive(double currentValue, double growthRate, int years){

if(years==0){

return currentValue;

}

return (1+growthRate) \* calculateFutureValueRecursive(currentValue, growthRate, years-1);

}

public static double calculateFutureValueMemo(double currentValue, double growthRate, int years, Map<Integer, Double> memo) {

if(years==0){

return currentValue;

}

if(memo.containsKey(years)){

return memo.get(years);}

double result=(1+growthRate) \* calculateFutureValueMemo(currentValue, growthRate, years- 1, memo);

memo.put(years, result);

return result;

}

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

System.out.print("Enter current value (e.g. investment amount): ");

double currentValue=sc.nextDouble();

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

double growthRate=sc.nextDouble();

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

int years=sc.nextInt();

long startRecursive=System.nanoTime();

double futureValueRecursive=calculateFutureValueRecursive(currentValue, growthRate, years);

long endRecursive=System.nanoTime();

long startMemo=System.nanoTime();

double futureValueMemo=calculateFutureValueMemo(currentValue, growthRate, years, new HashMap<>());

long endMemo=System.nanoTime();

System.out.printf("\nFinancial Forecast Results :\n");

System.out.printf("Future Value (Recursive) : %.2f\n", futureValueRecursive);

System.out.printf("Future Value (Memoized Recursive): %.2f\n", futureValueMemo);

System.out.println("\nTime Analysis (in nanoseconds) :");

System.out.println("Recursive Calculation Time : " + (endRecursive - startRecursive));

System.out.println("Memoized Recursive Time : " + (endMemo - startMemo));

System.out.println("\nComplexity Analysis :");

System.out.println("Recursive Version: Time Complexity -> O(n)");

System.out.println("Memoized Version : Time Complexity -> O(n) with reduced redundant calls");

System.out.println("\n6431499--Balaji V--CTS");

System.out.println("\nRecommendation...");

System.out.println("For small inputs, simple recursion is readable and effective.");

System.out.println("For larger forecasts, memoization ensures optimal performance by avoiding repeated computations.");

sc.close();

}

}

**Output:**  
