**Data Structures And Algorithm  
Mandatory**

**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**

**File: SearchProduct.java**

public class SearchProduct {

private int id;

private String name;

private String category;

public SearchProduct(int id, String name, String category) {

this.id = id;

this.name = name;

this.category = category;

}

public int getId() { return id; }

public String getName() { return name; }

public String getCategory() { return category; }

public String toString() {

return "Product[ID=" + id + ", Name=" + name + ", Category=" + category + "]";

}

}

**File: SearchAlgorithms.java**

public class SearchAlgorithms {

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

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

if (products[i].getId() == targetId) {

return i;

}

}

return -1;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

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

return mid;

}

if (products[mid].getId() < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

}

**File: SearchTest.java**

public class SearchTest {

public static void main(String[] args) {

SearchProduct[] products = {

new SearchProduct(1, "Phone", "Electronics"),

new SearchProduct(3, "Tablet", "Electronics"),

new SearchProduct(5, "Watch", "Accessories"),

new SearchProduct(7, "Headphones", "Audio")

};

int searchId = 5;

int linearResult = SearchAlgorithms.linearSearch(products, searchId);

int binaryResult = SearchAlgorithms.binarySearch(products, searchId);

System.out.println("Searching for Product ID: " + searchId);

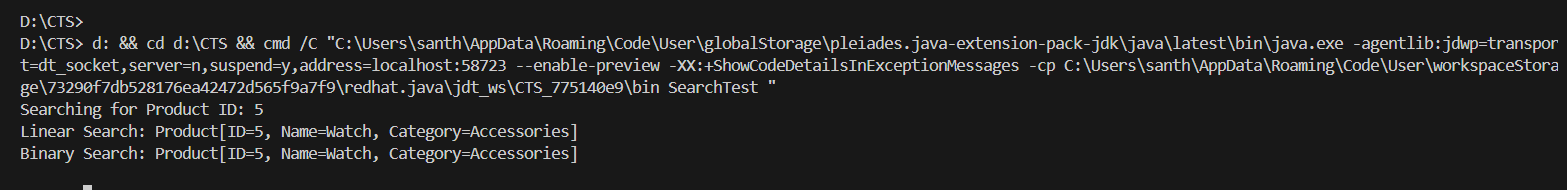
System.out.println("Linear Search: " + (linearResult != -1 ? products[linearResult] : "Not found"));

System.out.println("Binary Search: " + (binaryResult != -1 ? products[binaryResult] : "Not found"));

}

}

**Output:**



**Analysis:  
Linear Search:  
Best Case: O(1)  
Average Case:O(n)**

**Worst Case:O(n)**

**Binary Search:  
Best Case: O(1)  
Average Case:O(log n)**

**Worst Case:O(log n)**

**Conclusion:  
So,binary Search is best.**

**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**

**FinancialForecasting.java**

public class FinancialForecasting {

public static double calculateFutureValue(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue;

}

return calculateFutureValue(presentValue \* (1 + growthRate), growthRate, years - 1);

}

public static double calculateFutureValueIterative(double presentValue, double growthRate, int years) {

double futureValue = presentValue;

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

futureValue \*= (1 + growthRate);

}

return futureValue;

}

}

**ForecastingTest.java**

public class ForecastingTest {

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05;

int years = 10;

double recursiveResult = FinancialForecasting.calculateFutureValue(presentValue, growthRate, years);

double iterativeResult = FinancialForecasting.calculateFutureValueIterative(presentValue, growthRate, years);

System.out.println("Initial Investment: $" + presentValue);

System.out.println("Growth Rate: " + (growthRate \* 100) + "%");

System.out.println("Years: " + years);

System.out.println("Future Value (Recursive): $" + String.format("%.2f", recursiveResult));

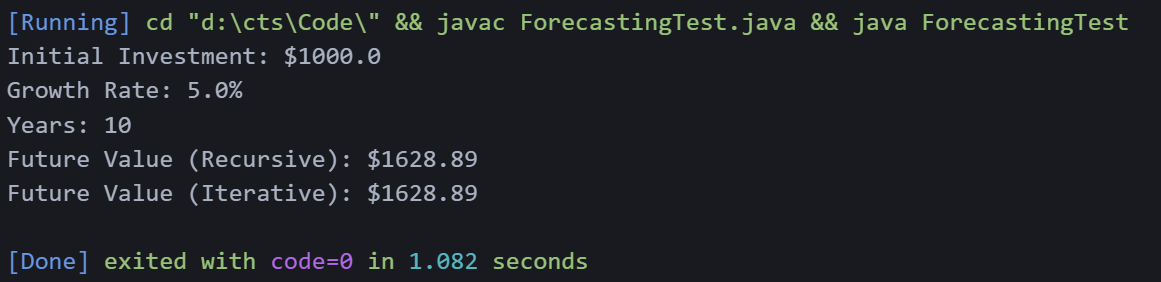
System.out.println("Future Value (Iterative): $" + String.format("%.2f", iterativeResult));

}

}

**Output:**

**Screenshots:**

****

**Analysis:**

**Time Complexity:  
Best Case:O(1)  
Average Case:O(n)  
Worst Case:O(n)**

**Optimization Strategies**

1. **Iterative Approach: Eliminates function call overhead**
2. **Memoization: Store computed results to avoid recomputation**
3. **Mathematical Formula: Use compound interest formula for O(1)**