Name:-Ritesh Kumar Sahoo

GitHub Link:- <https://github.com/Riteshksahoo/CognizantDotNet>CognizantDotNet

**-----------------------------------------------------------------------------------------**

**DATA STRUCTURE AND ALGORITHM  
  
Exercise 2: E-commerce Platform Search Function  
  
Understand Asymptotic Notation:**

* + **Explain Big O notation and how it helps in analyzing algorithms.**

Ans.:- Big O notation in data structures describes how the runtime or space requirements of an algorithm grow as the input size increases. This helps in understanding and comparing the efficiency of different algorithms.

Common Big O Notations:

1. Best Case (Ω): Fastest scenario
2. Average Case (Θ): Typical runtime for random inputs.
3. Worst Case (O): Maximum time the algorithm might take.
   * **Describe the best, average, and worst-case scenarios for search operations.**

Ans.:- Best Case Search

In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.

For linear search, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So the order of growth of time taken in terms of input size is constant.

Worst case Search

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.

For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one.

Average case Search

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.

**EcommerceSearch.java:-**package EcommerceSearch;

import java.util.Arrays;

import java.util.Comparator;

public class ECommerceSearch {

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

for (Product product : products) {

if (product.productId == targetId) {

return product;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

return products[mid];

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

left = mid + 1;

}

else {

right = mid - 1;

}

}

return null;

}

public static void sortProductsById(Product[] products) {

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

}

}

**Product.java:-**package EcommerceSearch;

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;

}

}

**Main.java:-**

package EcommerceSearch;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

new Product(105, "Shoes", "Fashion"),

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

new Product(104, "T-shirt", "Fashion")

};

int targetId = 102;

System.out.println("=== Linear Search ===");

Product result1 = ECommerceSearch.linearSearch(products, targetId);

printResult(result1);

ECommerceSearch.sortProductsById(products);

System.out.println("\n=== Binary Search ===");

Product result2 = ECommerceSearch.binarySearch(products, targetId);

printResult(result2);

}

private static void printResult(Product p) {

if (p != null) {

System.out.println("Found Product: " + p.productName +

" (ID: " + p.productId + ", Category: " + p.category + ")");

}

else {

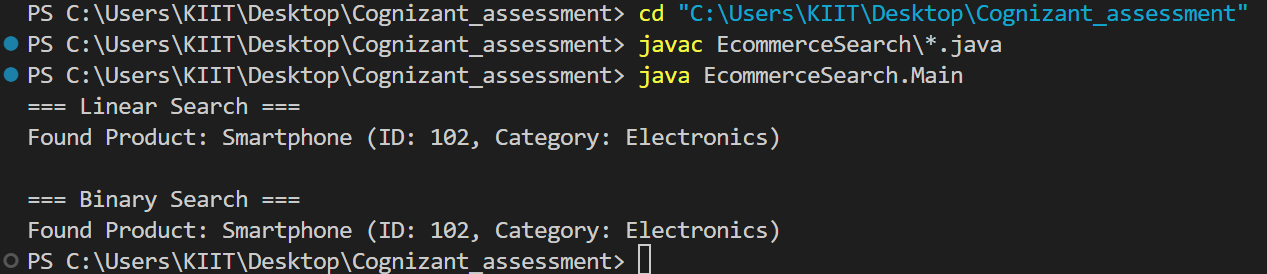
System.out.println("Product not found.");

}

}

}

**OUTPUT:-**



**Analysis:-**

1**. Compare the time complexity of linear and binary search algorithms.**

Ans.

|  |  |
| --- | --- |
| Search Algorithm | Time Complexities |
| Linear Search | O(n) |
| Binary Search | O(log n) |

1. **Discuss which algorithm is more suitable for your platform and why.**

Ans.:- Binary Search for large e-commerce platforms

**-----------------------------------------------------------------------------------------**

**Exercise 7: Financial Forecasting  
  
Understand Recursive Algorithms:**

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

Ans.:- Recursion is a programming technique where a function calls itself to solve a problem. It simplifies problems by breaking them down into smaller, self-similar subproblems. This approach is particularly effective for tasks involving repetitive patterns or hierarchical structures.   
It simplifies problems like:

* Fibonacci sequences
* Tree traversals
* Financial forecasts based on repeated growth

1. Recursion helps in logic building.
2. Recursive thinking helps in solving complex problems by breaking them into smaller subproblems.
3. Recursive solutions work as a a basis for Dynamic Programming and Divide and Conquer algorithms.  
     
    **FinancialForecasting.java:-**

import java.util.Scanner;

public class FinancialForecasting {

public static double calculateFutureValue(double initialValue, double rate, int years) {

if (years == 0) {

return initialValue;

}

return calculateFutureValue(initialValue, rate, years - 1) \* (1 + rate);

}

public static double calculateFutureValueMemo(double initialValue, double rate, int years, double[] memo) {

if (years == 0) {

return initialValue;

}

if (memo[years] != 0) {

return memo[years];

}

memo[years] = calculateFutureValueMemo(initialValue, rate, years - 1, memo) \* (1 + rate);

return memo[years];

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter Initial Investment Value: ");

double initialValue = sc.nextDouble();

System.out.print("Enter Annual Growth Rate (in %): ");

double ratePercent = sc.nextDouble();

System.out.print("Enter Number of Years: ");

int years = sc.nextInt();

double rate = ratePercent / 100.0;

double futureValue = calculateFutureValue(initialValue, rate, years);

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

double[] memo = new double[years + 1];

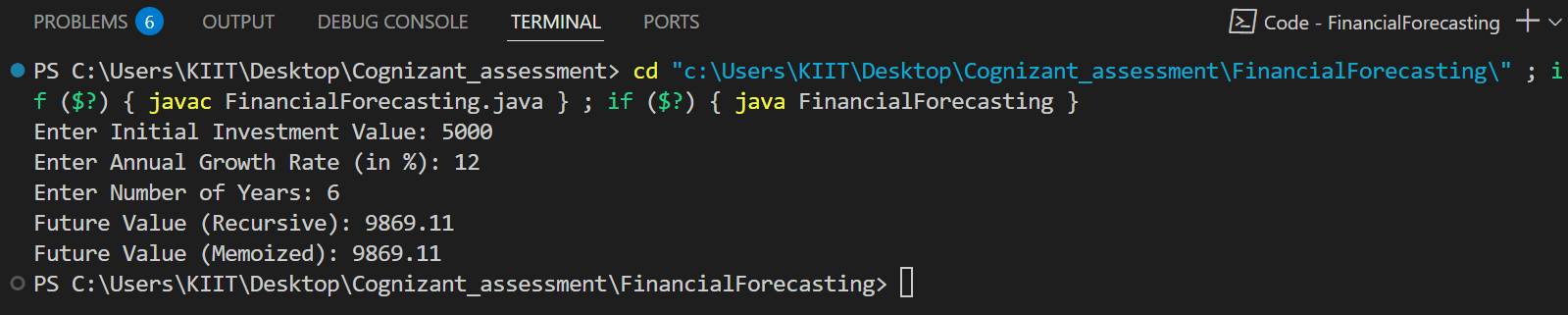
double futureValueMemo = calculateFutureValueMemo(initialValue, rate, years, memo);

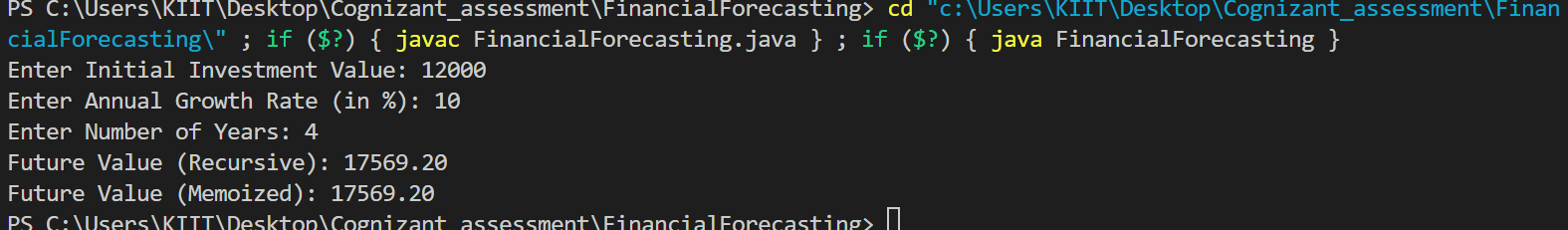
System.out.printf("Future Value (Memoized): %.2f\n", futureValueMemo);

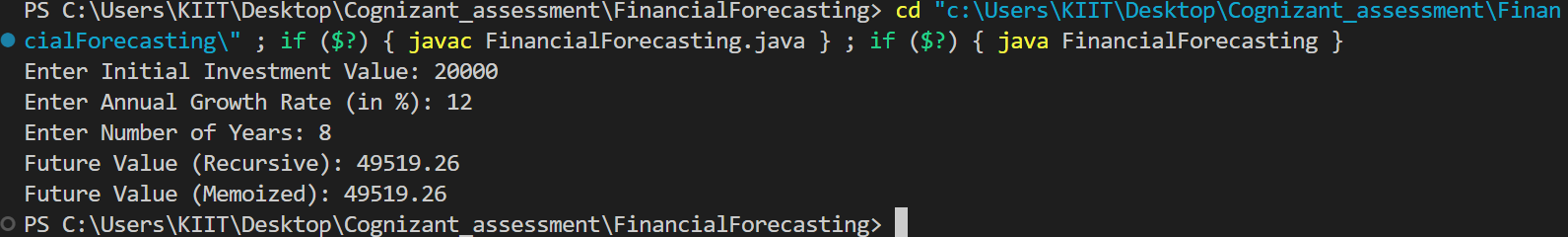
sc.close();

}

}

**OUTPUT:-  
1.**

**2**.

**3.**

**Analysis:-** 1**. Discuss the time complexity of your recursive algorithm.**

### Ans.:-Time Complexity

**T(n) = T(n-1) + O(1)** → **O(n)  
So, the time complexity is :- O(n)**

1. **Explain how to optimize the recursive solution to avoid excessive computation.**

Ans.:- We can optimize the recursive solution to avoid excessive computation.

Avoid simple recursion . Convert to Iterative i.e. Use Iteration Instead of Recursion :- It will avoid the stack overflow and reduce the space complexity from O(n) to O(1).