**Week 1**

**Data Structures and Algorithms**

**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.

Answer:

1. **Understanding Asymptotic Notation:**

Big O notation helps us to analyse and describe the efficiency of algorithms. It is a representation which indicates the time complexity of an algorithm relative to its input size. The ‘O’ in Big O stands for “order” while the value within the parentheses indicates the growth rate of the algorithm.

It implies that execution time of the algorithm increases proportionally with respect to the size of the input.

* Best, Average and Worst Cases in case of Search Operations:

**Best case** is when a function needs to perform the minimum number of operations on input data of n elements. It means the element that is being searched for is found at once.

**Average case** is when a function needs to perform an average number of steps on input data of n elements.

**Worst case** is when it requires a function to perform the maximum number of steps on input data of size n elements.

1. **Setup:**

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;

}

@Override

public String toString(){

return productId + " " + productName + " (" + category +")";

}

}

1. **Implementation (CODE):**

import java.util.\*;

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;

    }

    @Override

    public String toString() {

        return productId + " - " + productName + " (" + category + ")";

    }

}

public class ProductSearch {

    public static Product Linear\_Search(Product[] products, String target){

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

            Product curr\_product = products[i];

            if(curr\_product.productName.equalsIgnoreCase(target)){

                return curr\_product;

            }

        }

        return null;

    }

    public static Product Binary\_Search(Product[] products, String targetName) {

        int low = 0;

        int high = products.length - 1;

        while (low <= high) {

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

            Product midProduct = products[mid];

            int compare = midProduct.productName.compareToIgnoreCase(targetName);

            if (compare == 0) {

                return midProduct;

            } else if (compare < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] unsortedProducts = {

            new Product(001, "Watch", "Accessories"),

            new Product(003, "Bags", "Travel"),

            new Product(002, "Phone", "Electronics"),

            new Product(004, "Laptop", "Electronics"),

            new Product(005, "Home Decor", "Living")

        };

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

        Arrays.sort(sortedProducts, Comparator.comparing(p -> p.productName.toLowerCase()));

        Scanner scanner = new Scanner(System.in);

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

        String searchName = scanner.nextLine();

        Product resultLinear = Linear\_Search(unsortedProducts, searchName);

        System.out.println("Linear Search Result:");

        if (resultLinear != null) {

            System.out.println("Product Found: " + resultLinear);

        } else {

            System.out.println("Product Not Found.");

        }

        Product resultBinary = Binary\_Search(sortedProducts, searchName);

        System.out.println("Binary Search Result:");

        if (resultBinary != null) {

            System.out.println("Product Found: " + resultBinary);

        } else {

            System.out.println("Product Not Found.");

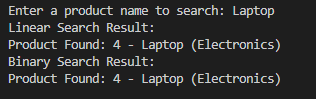
        }

        scanner.close();

    }

}

OUTPUT



1. **Analysis:**

* Time Complexity Comparison

|  |  |  |
| --- | --- | --- |
| Search Method | Time Complexity | Space Complexity |
| Linear Search | O(n) | O(1) |
| Binary Search | O(log n) | O(1) |

Since this is a search function for an e-commerce platform and it requires fast performance. Hence Binary Search is preferred over Linear Search.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

Answer:

**Understanding Recursive Algorithms:**

**Recursion** is the technique by which a function calls itself. It helps in breaking down complicated problems into simpler ones.

Example of Recursion:

Taking a complicated problem of adding a range of numbers together.

We can simplify this task by the help of Recursion.

Here, recursion is used to add the range of numbers together by breaking it down into a simple task of adding two numbers

public class Add {

public static void main(String[] args) {

int result = sum(25);

System.out.println(“The summation of the numbers is: ” + result);

}

public static int sum(int n){

int result;

// base condition

if(n == 1){

result = 1;

}else {

// recursive call

result = n + sum(n-1);

}

return result;

}

}

OUTPUT



* Working of Recursion in JAVA:

During recursive call, whenever a method calls itself, its entry is pushed to the stack till the base condition terminates the flow. When base condition comes true, and method starts returning the value, the result of sub call is popped from the stack and so on till the all entries of method is popped from the stack.

**Base Condition** is the stopping point or the termination point where the recursive calls cease and the function returns a value. It prevents the recursion functions to run into the problem of infinite recursion.

**Setup and Implementation**

This method is created to predict the future value based on past data using the Simple Interest Formula. It uses recursion for the future value prediction.

public class FinancialForecasting {

public static double future\_val(double principal, double rate, int years){

if (years == 0){

return principal;

}

return future\_val(principal,rate,years - 1) + (principal \* rate);

}

public static void main(String[] args){

double principal = 1000.0;

double rate = 0.12;

int years = 10;

double res = future\_val(principal, rate, years);

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

}

}

OUTPUT



In this approach:

Time Complexity:

O(n) due to 1 recursive call for each year

Space Complexity: O(n)

Optimization to avoid extra recursive calls:

In order to optimize the iterative approach is used to directly compute the future value

It takes:

Time Complexity: O(1)

Space Complexity: O(1)

public class FinancialForecasting {

public static double future\_val(double principal, double rate, int years){

if (years == 0){

return principal;

}

return principal + (principal \* rate \* years);

}

public static void main(String[] args){

double principal = 1000.0;

double rate = 0.12;

int years = 10;

double res = future\_val(principal, rate, years);

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

}

}

OUTPUT

