**SOLUTION WEEK 1**

**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.

EXPLAINATIONS:

Big O notation is a way to describe the upper bound of an algorithm's running time or space usage as the input size grows, focusing on the growth rate rather than specific execution times. It helps analyze algorithm efficiency by providing a standardized way to compare how different algorithms scale with increasing input. It allows for a standardized comparison of algorithms, abstracting away implementation details and focusing on the inherent scalability and efficiency. An algorithm with a time complexity of O(n) will take roughly proportional time to the input size (n), while an algorithm with O(n2) will take time proportional to the square of the input size.

For linear search:

Best Case: Constant Time irrespective of input size. This will take place if the element to be searched is on the first index of the given list. So, the number of comparisons, in this case, is 1.  
Average Case: Linear Time, this will take place if the element to be searched is at the middle index (in an average search) of the given list.  
Worst Case: The element to be searched is not present in the list

For binary search:

Best case: where the target value is the middle element of the array, its position is returned after one iteration.

Worst case: may also be reached when the target element is not in the array.

CODE

public class Product {

    public int productid;

    String category;

    String  productname;

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

        this.productid=id;

        this.category=category;

        this.productname=name;

    }

}

import java.util.Scanner;

public class Search {

  public static void main(String[] args) {

    Product[] products = new Product[7];

    products[0] = new Product(1,"tshirt","clothes");

    products[1] = new Product(2,"jeans","clothes");

    products[2] = new Product(3,"mixer","electronics");

    products[3] = new Product(4,"PC","electronics");

    products[4] = new Product(5,"pillow cover","home decor");

    products[5] = new Product(6,"lamps","home decor");

    products[6] = new Product(7,"notebooks","stationery");

    Scanner scanner =new Scanner(System.in);

    System.out.print("Enter the search value: ");

    int value = scanner.nextInt( );

    System.out.print("Choose the search method (1 for linear search),(2 for binary search) ");

    int ch = scanner.nextInt( );

    int foundIndex = -1;

    if(ch==1){

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

    if(products[j].productid == value) {

      foundIndex = j;

      break;

    }

    }

    }

     else{

      int l=0,h=products.length-1;

      while (l<=h) {

        int m=(l+h)/2;

        if (products[m].productid == value){

          foundIndex=m;

          break;

        }

        else if(products[m].productid > value)

          h=m-1;

        else

          l=m+1;

     }

  }

  if (foundIndex > -1){

  System.out.println("Found at " + foundIndex);

  System.out.println("Category " + products[foundIndex].category);

  System.out.println("Product Name" + products[foundIndex].productname);

}

else

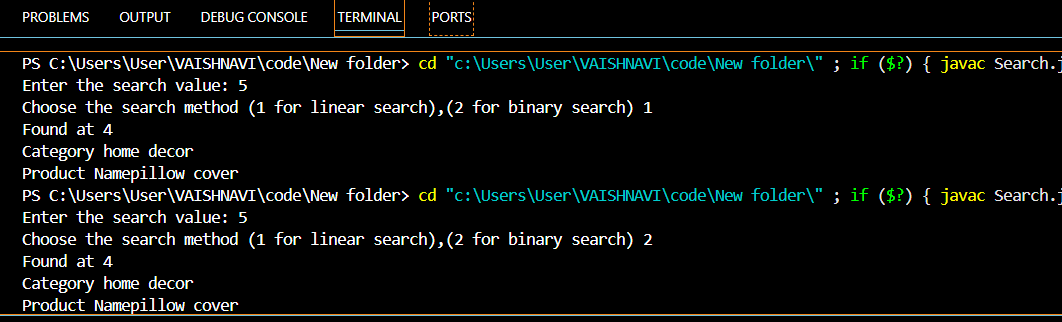
  System.out.println("Not found.");

  scanner.close();

  }

}

OUTPUT



|  |  |
| --- | --- |
| **Linear Search** | **Binary Search** |
| **The time complexity of linear search O(n).** | **The time complexity of binary search O(log n).** |
| **Linear search performs equality comparisons** | **Binary search performs ordering comparisons** |
| **It is less complex.** | **It is more complex.** |
| **It is very slow process.** | **It is very fast process.** |

For this e commerce platform binary search is better as compared to linear search because binary search is faster and performs less comparisons than linear search **.**A e commerce platform may have a large amount of product to search so linear search is not a good option.

**Exercise 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.

EXPLAINATION:

Recursion is a programming technique where a function calls itself within its own definition. It's a powerful way to solve problems that can be broken down into smaller, self-similar subproblems. Recursion simplifies complex problems by reducing them to simpler instances of the same problem. A key characteristic is the presence of a base case, which stops the recursion and provides a known starting point.

CODE

import java.util.Scanner;

public class prediction {

    public static void main(String[] args){

        double past\_value,growth\_rate;

        int period;

        Scanner sc =new Scanner(System.in);

        System.out.println("Enter past value:");

        past\_value=sc.nextDouble();

        System.out.println("Enter growth rate:");

        growth\_rate=sc.nextDouble();

        System.out.println("Enter number of periods:");

        period=sc.nextInt();

        double future\_value=calculate\_future\_value(past\_value,growth\_rate,period);

        System.out.println("Future value is:"+future\_value);

        sc.close();

    }

    public static double calculate\_future\_value(double pv,double gv,int p){

        if(p==0){

            return pv;

        }

        else{

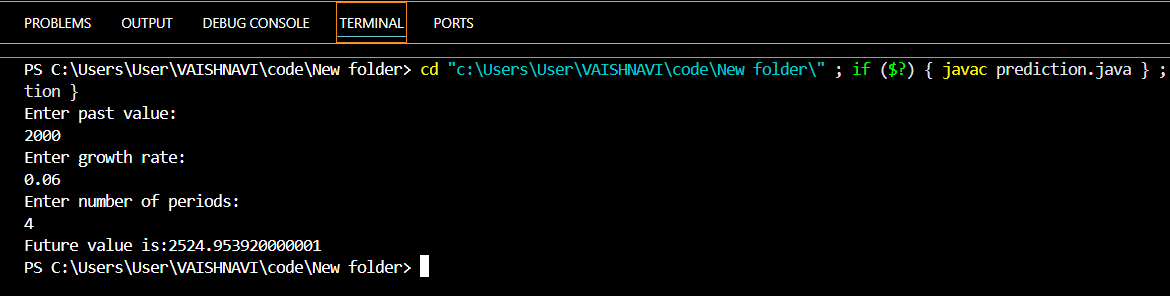
            return calculate\_future\_value(pv\*(1+gv), gv, p-1);

        }

    }

}

OUTPUT



The time complexity of this recursive algorithm is O(n), where n is the number of periods, because the function calls itself n times in the worst-case scenario.

The recursive solution, is simple, can be inefficient for large values of 'n' due to the overhead of function calls. A more efficient approach is to use an iterative solution.

The iterative solution has a time complexity of O(n) but avoids the overhead of recursive function calls, making it faster for larger values of 'n'. In general, iterative solutions are preferred over recursive solutions for performance reasons when the recursion depth becomes significant.