Algorithms and Data Structures

**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:

i)Understanding Asymptotic Notation:

**Big O Notation** describes how the runtime of an algorithm grows with input size:

* **Best Case**: Minimum time (e.g., first element match in search)
* **Average Case**: Expected time over all inputs
* **Worst Case**: Maximum time (e.g., element not found)

For example:

* **Linear Search**: Best O(1), Average/Worst O(n)
* **Binary Search**: Best O(1), Average/Worst O(log n)

**2. Setup**

Create a Product class with searchable attributes:

import java.util.Arrays;

import java.util.Comparator;

import java.util.\*;

class Product {

    int productId;

    String productName;

    String category;

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

        this.productId = id;

        this.productName = name;

        this.category = category;

    }

}

**3. Implementation**

**Linear Search** (unsorted array):

public class SearchDemo{

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

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

            if(products[i].productName.equalsIgnoreCase(target)){

                return i;

            }

        }

        return -1;

    }

**Binary Search** (sorted array):

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

        int left=0,right=products.length-1;

        while(left<=right){

            int mid=(left+right)/2;

            int cmp=products[mid].productName.compareToIgnoreCase(target);

            if(cmp==0) return mid;

            else if(cmp<0) left=mid+1;

            else right=mid-1;

        }

        return -1;

    }

**Main function:**

public static void main(String args[]){

        Scanner in=new Scanner(System.in);

        System.out.print("Enter number of Products");

        int n=in.nextInt();

        in.nextLine();

        Product[] products=new Product[n];

        for(int i=0;i<n;i++){

            System.out.println("Enter details for product "+(i+1));

            System.out.print("Product ID: ");

            int productId=in.nextInt();

            in.nextLine();

            System.out.print("Product Name: ");

            String productName=in.nextLine();

            System.out.print("Category: ");

            String productCategory=in.nextLine();

            products[i]=new Product(productId,productName,productCategory);

        }

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

        String target=in.nextLine();

        int linearResult=linearSearch(products,target);

        System.out.println("Linear Search result index: "+linearResult);

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

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

        int binaryResult=binarySearch(sortedProducts,target);

        System.out.println("Binary Search result index(after sorting):"+binaryResult);

        in.close();

    }

}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**4)Analysis:**

**i)Comparing time complexities of linear and binary search:**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

* **Linear Search** checks every element one by one.
  + Best when the element is at the beginning.
  + Worst when it’s at the end or missing entirely.
* **Binary Search** works only on a sorted list.
* Splits the array in half each time.
* Much more efficient for larger datasets.

->Which Is More Suitable for an E-Commerce Platform?

For an e-commerce platform, **binary search** is usually the better fit — and here’s why:

* **Performance at Scale**: As the number of products grows into the thousands or millions, binary search remains lightning-fast, while linear search slows dramatically.
* **Sorted Data**: Product listings can be maintained in sorted order (by name, price, etc.) or indexed efficiently.
* **User Experience**: Faster search = faster results = happier customers.

However, linear search may still be useful when:

* The dataset is very small.
* The search needs to be flexible across multiple fields (like checking both name and category).
* The data changes constantly and sorting on every update isn't feasible.

**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.

**Solution:**

1. **Understand Recursive Algorithms:**

**Recursion:** Recursion is when a method calls itself to solve a smaller version of the same problem, gradually working its way down to a simple case — called the base case — where it can stop.

In forecasting problem, **the base case would be when years==0.**

And it recursively calculates the amount until it reaches the base case, when it recaches the base case , it stops and return the amount.

The recursive approach would be like:

**forecast(years) = forecast(years - 1) × (1 + growthRate)**

**2)Setup:**

**A recursive function:**

public class Forecast {

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

        if (years == 0) return currentValue;

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

    }

**3)Implementation**

import java.util.\*;

public class Forecast {

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

        if (years == 0) return currentValue;

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

    }

    public static void main(String[] args) {

        Scanner in=new Scanner(System.in);

        System.out.print("Enter current value: ");

        double currentValue=in.nextInt();

        System.out.print("Enter annual growth rate(as a percentage for ex:5):");

        double growthPercent=in.nextDouble();

        double growthRate=growthPercent/100.0;

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

        int years=in.nextInt();

        double result=forecastValue(currentValue,growthRate,years);

        System.out.printf("Predicted value after %d years: Rs %.2f%n",years,result);

        in.close();

    }

}

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**4)Analysis:**

The recursive approach has a complexity which is O(n), where n is the amount of years:

We are reducing years by 1 in each recursive call.

So for n years, it calls itself n times before you get to a base case.

There’s no branching or overlapping subproblems at all, so it’s definitely linear.

The space complexity is also O(n) because of implicit call stack — all recursive calls go at the stack: as soon as we reach base case, we can easily pop all the elements at the stack**.**

**Optimize recursive solution to avoid excessive computation:**

**1. Use an Iterative Approach**

**Instead of recursion, use a loop:**

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

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}

Analysis: Same time complexity: **O(n)**

* But **space complexity drops to O(1)** — no call stack buildup
* More robust for large n (e.g., 10,000 years)