

# **ALGORITHMS 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:**

### **1. Understand Asymptotic Notation:**

- Big O notation is a powerful tool used to describe the time complexity or space complexity of algorithms. Big-O is a way to express the upper bound of an algorithm's time or space complexity. It describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value. It can be used to compare the efficiency of different algorithms or data structures. It provides an upper limit on the time taken by an algorithm in terms of the size of the input. We mainly consider the worst case scenario of the algorithm to find its time complexity in terms of Big O. It's denoted as  $O(f(n))$ .  
It helps in analyzing algorithms by providing a way to describe how the runtime or space requirements of an algorithm grow as the input size increases. It allows programmers to compare different algorithms and choose the most efficient one for a specific problem.
- In linear search, which is used on unsorted arrays or lists, the best-case scenario occurs when the target element is found at the very beginning of the array. In this case, only one comparison is needed, resulting in a time complexity of  $O(1)$ . The average-case scenario assumes that the target element is somewhere in the middle of the list. On average, the search will check about half the elements, which leads to a time complexity of  $O(n)$ . The worst-case scenario happens when the target element is either at

the end of the list or not present at all. This means every element must be checked, giving a time complexity of  $O(n)$ .

In binary search, which operates on sorted arrays, the best-case scenario occurs when the middle element of the array is the target, found on the first comparison. This results in a time complexity of  $O(1)$ . The average-case scenario involves repeatedly halving the array and checking the middle element, typically requiring  $\log n$  comparisons, giving a time complexity of  $O(\log n)$ . The worst-case scenario also requires  $O(\log n)$  time when the element is either not present or found after many splits.

#### Setup and Implementation:

```
import java.util.*;

class Product implements Comparable<Product> {
    int productId;
    String productName;
    String category;
    public Product(int productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName.toLowerCase();
        this.category = category.toLowerCase();
    }
    public int compareTo(Product other) {
        return this.productName.compareTo(other.productName);
    }
    public String toString() {
        return "ProductID: " + productId + ", Name: " + productName + ", Category: " + category;
    }
}

public class EcommerceSearchInput {
    public static Product linearSearch(Product[] products, String targetName) {
        for (Product product : products) {
            if (product.productName.equalsIgnoreCase(targetName)) {
                return product;
            }
        }
        return null;
    }
    public static Product binarySearch(Product[] products, String targetName) {
        int left = 0, right = products.length - 1;
        targetName = targetName.toLowerCase();
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int cmp = products[mid].productName.compareTo(targetName);
            if (cmp == 0)
                return products[mid];
            else if (cmp < 0)
                left = mid + 1;
```

```

        else
            right = mid - 1;
    }
    return null;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter number of products: ");
    int n = sc.nextInt();
    sc.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 id = sc.nextInt();
        sc.nextLine();
        System.out.print("Product Name: ");
        String name = sc.nextLine();
        System.out.print("Category: ");
        String category = sc.nextLine();
        products[i] = new Product(id, name, category);
    }
    System.out.print("\nEnter product name to search: ");
    String searchName = sc.nextLine();
    Product foundLinear = linearSearch(products, searchName);
    System.out.println("\nLinear Search Result:");
    System.out.println(foundLinear != null ? foundLinear : "Product not found");
    Arrays.sort(products);
    Product foundBinary = binarySearch(products, searchName);
    System.out.println("\nBinary Search Result:");
    System.out.println(foundBinary != null ? foundBinary : "Product not found");
    sc.close();
}
}

```

## OUTPUT:

```
Enter number of products: 3
Enter details for product 1
Product ID: 567
Product Name: soap
Category: washing clothes
Enter details for product 2
Product ID: 234
Product Name: toy
Category: doll
Enter details for product 3
Product ID: 120
Product Name: dress
Category: gown

Enter product name to search: toy

Linear Search Result:
ProductID: 234, Name: toy, Category: doll

Binary Search Result:
ProductID: 234, Name: toy, Category: doll
```

### 4. Analysis:

- Linear search checks each element one by one and works on both sorted and unsorted data. Its time complexity is  $O(n)$  in the worst case, making it slower for large datasets. Binary search, however, works only on sorted data. It divides the search range in half repeatedly, which significantly reduces the number of comparisons. Its time complexity is  $O(\log n)$  in the worst case, making it much faster than linear search for large, sorted datasets.
- For an e-commerce platform, binary search is generally more suitable than linear search due to its significantly better performance and scalability. Binary search has a time complexity of  $O(\log n)$ , which makes it highly efficient for searching large, sorted datasets—common in real-world e-commerce systems where thousands or even millions of products may be listed. Binary search works best when the data is already sorted, which is often the case in platforms where products are organized alphabetically, by ID, or price. This organization allows for quicker lookups and an improved user experience.

## **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 is a programming technique in which a function calls itself to solve smaller instances of the same problem. It works by breaking down a complex problem into simpler sub-problems, each of which is solved using the same recursive logic. Every recursive function must have a base case to stop the recursion and prevent infinite loops. This approach is particularly useful for solving problems that have a natural hierarchical or repetitive structure, such as calculating factorials, generating Fibonacci numbers, traversing trees or graphs, and solving mathematical puzzles like the Tower of Hanoi. Recursion often makes the code more elegant and easier to understand compared to iterative solutions, especially when dealing with nested or self-similar data. However, care must be taken to ensure the recursion terminates correctly, as excessive or uncontrolled recursion can lead to stack overflow errors. Overall, recursion is a powerful tool that simplifies certain problems by reducing them to smaller, more manageable versions of themselves.

### **Setup and Implementation:**

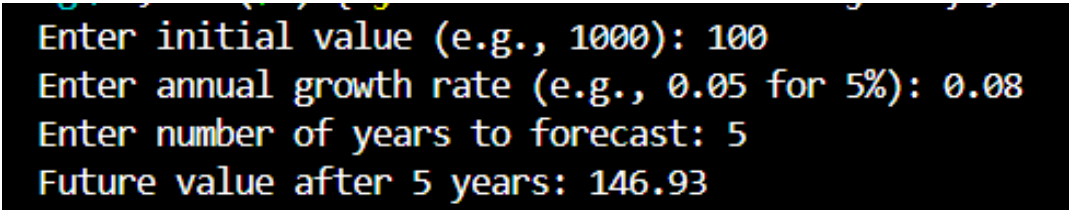
```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
public class FinancialForecast {
    private static Map<Integer, Double> memo = new HashMap<>();
    public static double forecastValue(double initialValue, double growthRate, int years) {
        if (years == 0) {
            return initialValue;
        }
        if (memo.containsKey(years)) {
            return memo.get(years);
        }
    }
}
```

```

    }
    double futureValue = forecastValue(initialValue, growthRate, years - 1) * (1 + growthRate);
    memo.put(years, futureValue);
    return futureValue;
}
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter initial value (e.g., 1000): ");
    double initialValue = sc.nextDouble();
    System.out.print("Enter annual growth rate (e.g., 0.05 for 5%): ");
    double growthRate = sc.nextDouble();
    System.out.print("Enter number of years to forecast: ");
    int years = sc.nextInt();
    double result = forecastValue(initialValue, growthRate, years);
    System.out.printf("Future value after %d years: %.2f\n", years, result);
    sc.close();
}
}

```

### **OUTPUT:**



```

Enter initial value (e.g., 1000): 100
Enter annual growth rate (e.g., 0.05 for 5%): 0.08
Enter number of years to forecast: 5
Future value after 5 years: 146.93

```

#### **4. Analysis:**

- The time complexity of the recursive financial forecasting algorithm depends on whether memoization is used. In the basic recursive version without memoization, the function makes a single recursive call for each year, decreasing the year count by one until it reaches zero. This results in a linear time complexity of  $O(n)$ , where  $n$  is the number of years to forecast. However, this version can be inefficient for large inputs due to repeated calculations and increased stack usage. To optimize the solution, memoization is introduced by storing previously computed results in a map. With memoization, each recursive call for a unique year is computed only once and retrieved in constant time thereafter. This reduces redundant calculations and maintains the time complexity at  $O(n)$ , while significantly improving efficiency. The space complexity also becomes  $O(n)$  due to storage in both the memoization map and the call stack. For scenarios where stack memory is a concern, an iterative approach can be adopted, which also runs in  $O(n)$  time but with  $O(1)$  space complexity, making it the most efficient in practice.

- To optimize the recursive solution and avoid excessive computation, memoization can be used. Memoization is a technique that stores the results of expensive function calls and reuses them when the same inputs occur again. In the context of financial forecasting, each recursive call computes the future value for a specific year based on the previous year's value. Without memoization, the algorithm recalculates the same intermediate values multiple times, leading to unnecessary computations and increased execution time. By storing the computed future values for each year in a data structure like a HashMap, the algorithm can directly retrieve the result for any previously computed year in constant time. This significantly reduces redundant calculations and improves efficiency. Additionally, for even better performance and to avoid potential stack overflow in case of very large inputs, the recursive approach can be replaced with an iterative solution, which eliminates recursion altogether and reduces the space complexity to constant space.

NAME-RITIKA KUMARI

SUPERSET ID- 6392654