**DATA STRUCTURES AND ALGORITHMS HANDS-ON**

**Exercise 2: E-commerce Platform Search Function**

### Understanding Asymptotic Notation

Asymptotic notation, particularly Big O notation, is used to describe how the performance of an algorithm changes as the size of the input increases. It helps in estimating the upper bound of time or space complexity in the worst-case scenario, making it easier to compare different algorithms.

#### Common Big O Notations:

* O(1): Represents constant time, where the execution time remains the same regardless of the input size.
* O(n): Indicates linear time, where the execution time increases directly with the input size.
* O(log n): Refers to logarithmic time, which is much faster and commonly seen in algorithms like binary search.

### Best, Average, and Worst-Case Analysis

* Linear Search:
  + Best case: The element is found at the beginning, requiring only one comparison (O(1)).
  + Worst case: The element is found at the end or not present, requiring a full scan through the input (O(n)).
  + Average case: The search may go through about half the elements on average (O(n)).
* Binary Search:
  + Best case: The target element is right in the middle of the data (O(1)).
  + Worst and average cases: The search space is halved with each step, leading to logarithmic time complexity (O(log n)).

Code:

import java.util.Arrays;

import java.util.Comparator;

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;

}

public String toString() {

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

}

}

public class SearchDemo {

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

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

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

return i;

}

}

return -1;

}

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

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

while (left <= right) {

int mid = (left + right) / 2;

if (products[mid].productId == targetId) return mid;

else if (products[mid].productId < targetId) left = mid + 1;

else right = mid - 1;

}

return -1;

public static void main(String[] args) {

Product[] products = {

new Product(103, "Shoes", "Footwear"),

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

new Product(105, "Watch", "Accessories"),

new Product(102, "T-Shirt", "Apparel"),

new Product(104, "Laptop", "Electronics")

};

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

int index1 = linearSearch(products, 104);

System.out.println(index1 != -1 ? "Found: " + products[index1] : "Not Found");

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

System.out.println("\nBinary Search (sorted by productId):");

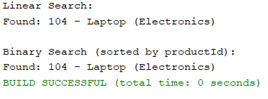
int index2 = binarySearch(products, 104);

System.out.println(index2 != -1 ? "Found: " + products[index2] : "Not Found");

}

}

**Output:**



### Implementation Approach

In a linear search, each product in the list is checked one after the other until the desired item is found. This method doesn’t require the data to be sorted and can work on any list.

Binary search, on the other hand, is suitable only for sorted lists. It begins by examining the middle item and then decides whether to search the left or right half based on whether the target product is smaller or larger. This approach significantly cuts down the number of comparisons needed to find an item.

To use binary search, the product list must first be arranged in alphabetical order or sorted in some consistent way.

### Time Complexity Analysis

The time complexity of linear search is O(n), meaning it may need to go through every item in the list. While easy to implement, it becomes inefficient for large amounts of data.

Binary search has a time complexity of O(log n), which makes it far more efficient for larger datasets. However, it requires the data to be sorted in advance to work correctly.

**Exercise 7: Financial Forecasting**

Recursion Concept

Recursion is a technique where a method calls itself to solve smaller instances of the same problem.It’s useful for problems like factorials, tree traversal, or repetitive calculations like financial projections over years.

Code:

public class FinancialForecast {

public static double predictFutureValue(double currentValue, double annualGrowthRate, int years) {

if (years == 0) {

return currentValue; }

return predictFutureValue(currentValue \* (1 + annualGrowthRate), annualGrowthRate, years - 1);

}

public static double predictFutureValueMemo(double currentValue, double annualGrowthRate, int years, double[] memo) {

if (memo[years] != 0) return memo[years];

if (years == 0) return currentValue;

memo[years] = predictFutureValueMemo(currentValue \* (1 + annualGrowthRate), annualGrowthRate, years - 1, memo);

return memo[years];

}

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 0.08;

int years = 5;

double future = predictFutureValue(initialValue, growthRate, years);

System.out.println("Recursive Prediction: " + future);

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

double optimizedFuture = predictFutureValueMemo(initialValue, growthRate, years, memo);

System.out.println("Optimized Recursive Prediction: " + optimizedFuture);

}

}

**Output:**



### **Analysis**

* The recursive function **predictFutureValue** has a time complexity of O(n), where n represents the number of years being projected. This is because the function reduces the problem by one year with each recursive call until it reaches the base case. Since each call involves a fixed amount of computation, the total time taken grows linearly with the number of years.
* When recursion is used for forecasting, it can lead to repeated calculations of the same values, especially in more advanced cases. To handle this, memoization can be applied. Memoization involves saving the result of a subproblem the first time it’s solved, allowing future calls to simply look up the answer instead of recalculating it. This greatly improves efficiency, particularly in more complex forecasting situations with multiple growth rates or decision branches.