# Data structures and Algorithms

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

**1. Understanding Asymptotic Notation**

**Big O Notation** is used to describe the time complexity of algorithms, giving insight into how an algorithm performs as the input size increases. It helps in selecting the most efficient algorithm for a task, especially in performance-critical applications like e-commerce.

In the context of search:

* **Linear Search** has:
  + Best Case: **O(1)** – when the element is at the beginning.
  + Average and Worst Case: **O(n)** – where n is the number of products.
* **Binary Search** has:
  + Best Case: **O(1)** – when the element is in the middle.
  + Average and Worst Case: **O(log n)** – efficient for large, sorted datasets.

**Product Class**

public 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 + ")";

}

}

**3. Implementation**

**Linear Search**: This method goes through each product in the array and checks for a match.

public class LinearSearch {

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

for (Product product : products) {

if (product.productId == targetId) {

return product;

}

}

return null;

}

}

**Binary Search**: This method works on sorted arrays and uses a divide-and-conquer approach for faster search.

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static Product search(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 products[mid];

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

else right = mid - 1;

}

return null;

}

public static void sortProducts(Product[] products) {

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

}

}

**Test Class**

public class SearchTest {

public static void main(String[] args) {

Product[] products = {

new Product(104, "Shoes", "Footwear"),

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

new Product(103, "Chair", "Furniture"),

new Product(102, "Phone", "Electronics")

};

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

Product result1 = LinearSearch.search(products, 103);

System.out.println(result1 != null ? result1 : "Product not found");

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

BinarySearch.sortProducts(products);

Product result2 = BinarySearch.search(products, 103);

System.out.println(result2 != null ? result2 : "Product not found");

}

}

**5. Analysis**

Linear search is simple and does not require the data to be sorted. It is suitable for smaller datasets or when the data changes frequently and maintaining a sorted structure would be inefficient. However, as the dataset grows, its performance declines due to its linear time complexity.

Binary search, in contrast, is highly efficient for large datasets, but it requires the array to be sorted. It significantly reduces the number of comparisons needed to find an item, thanks to its logarithmic time complexity. In an e-commerce platform where users expect quick results and the product catalog is large and mostly static, binary search is the better choice. It provides faster response times and ensures a better user experience, especially when searching by product IDs or other numeric keys.

**Exercise 7: Financial Forecasting**

**1. Understanding Recursive Algorithms**

* Recursion is a technique where a method calls itself.
* It simplifies problems by breaking them into smaller subproblems.
* Useful for problems like financial forecasting where values build on previous ones.

**2. Setup**

* Method to calculate future value: FV(n) = FV(n-1) \* (1 + growthRate)
* Base case: FV(0) = currentValue

**3.Implementation**  
public class FinancialForecast {

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

if (years == 0) {

return currentValue;

} else {

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

}

}

public static void main(String[] args) {

double currentValue = 1000.0;

double growthRate = 0.08;

int years = 5;

double predictedValue = futureValue(years, currentValue, growthRate);

System.out.println("Predicted value after " + years + " years: " + predictedValue);

}

}

**4. Analysis**

The time complexity of the recursive function is **O(n)**, where n is the number of years. This is because the function makes one recursive call for each year until it reaches the base case.

Although the current implementation is efficient for small input sizes, recursion can become inefficient if the same subproblems are recomputed repeatedly (as in naive Fibonacci calculations). However, in this forecasting scenario, since each year depends directly on the previous one and not on multiple sub-values, the recursion depth and computation are linear and manageable.

To optimize further and avoid excessive computation for very large values of n, we could:

* Use memoization to store and reuse already calculated values.
* Use an iterative approach, which is generally more efficient and avoids the risk of stack overflow in deep recursive calls.

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

double value = currentValue;

for (int i = 1; i <= years; i++) {

value \*= (1 + growthRate);

}

return value;