**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.

**Asymptotic Notations:**

Asymptotic Notation is a mathematical method used to describe the **efficiency** of an algorithm when the size of the input data becomes very large. It provides a way to express the **growth of time or space requirements** of an algorithm relative to the input size, usually denoted as n.

**BIG O:**

Big O Notation (written as O(f(n))) specifically represents the upper bound, or worst-case growth rate, of an algorithm. It describes how quickly the execution time or memory usage increases as the input size increases.

For example:

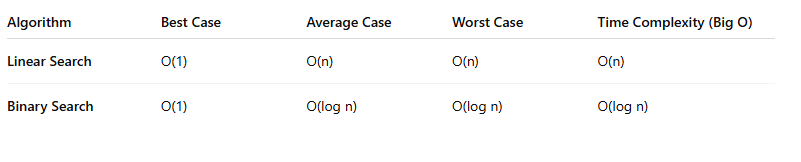
O(1) → Constant time (doesn’t grow with input size)

O(n) → Linear time (time grows proportionally with input size)

O(log n) → Logarithmic time (time grows slowly, even for large inputs)

Big O simplifies performance comparisons by ignoring machine-specific details and focusing only on algorithmic growth.

* Helps developers predict how an algorithm will perform when the input data increases.
* Makes it easier to compare multiple algorithms solving the same problem.
* Guides the choice of algorithm in real-world applications — for example, optimizing search functionality in an e-commerce platform.



**CODE:**

**Product.java**

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;

}

@Override

public String toString() {

return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**SearchDemo.java**

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(productName)) {

return product;

}

}

return null;

}

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

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

while (left <= right) {

int mid = left + (right - left) / 2;

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

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shirt", "Clothing"),

new Product(103, "Book", "Education"),

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

new Product(105, "Shoes", "Footwear")

};

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

String searchName = "Laptop";

Product linearResult = linearSearch(products, searchName);

System.out.println("Linear Search Result: " + (linearResult != null ? linearResult : "Product not found."));

Product binaryResult = binarySearch(products, searchName);

System.out.println("Binary Search Result: " + (binaryResult != null ? binaryResult : "Product not found."));

}

}

**ANALYSIS:**

**Linear search** has a time complexity of **O(n)**, meaning it may need to check every element in the list to find a match. It works on **unsorted data** and is **simple to implement**, but becomes **inefficient** as data size grows. **Best case** is **O(1)** if the item is first, but the **average and worst case** are **O(n)**.

**Binary search**, on the other hand, works only on **sorted data** and has a much faster time complexity of **O(log n)**. It repeatedly divides the dataset in half, quickly narrowing down the search. Its **best, average, and worst cases** are **O(1)** and **O(log n)** respectively. It’s ideal for **large datasets**.

For an **e-commerce platform** where product data is **large**, **binary search is the better choice** because it is **faster** and **more scalable**. Modern databases and search engines often use **indexes** and **search trees** based on this principle to deliver quick results.

**Conclusion**:

Linear search = good for **small/unsorted** datasets where as Binary search = **recommended** for **large, sorted** datasets in real-world systems.

**Output:**



**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.

**RECURSION:**

**Recursion** is a programming concept where a function **calls itself** to solve smaller instances of the same problem. Each recursive call works on a smaller input, gradually moving toward a **base case** — a condition that stops the recursion. Once the base case is reached, the function starts returning values back through the chain of calls. Recursion is especially useful for problems that have a **repeating structure**, such as mathematical sequences, tree structures, or when a problem can naturally be broken down into smaller, similar subproblems.

One of the main advantages of recursion is that it allows developers to write **simpler and more readable code** for problems that would otherwise require complex looping or nested structures. Examples of problems often solved with recursion include **calculating factorials**, **finding Fibonacci numbers**, and **traversing hierarchical data structures** like file systems or organizational charts. In the context of **financial forecasting**, recursion can be used to repeatedly apply growth rates over multiple periods, making the code logically match the way such forecasts are structured in real life.

In summary, recursion **simplifies complex problems** by breaking them down into smaller, manageable tasks — each solved by the function itself.

**CODE:**

**FinancialForecast.java**

public class FinancialForecast {

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

double presentValue = 10000;

double growthRate = 0.08;

int years = 5;

double result = futureValue(presentValue, growthRate, years);

System.out.printf("Future Value after %d years: ₹%.2f\n", years, result);

}

}

**ANALYSIS:**

The time complexity of this recursive algorithm is O(n), where n is the number of years. This is because for each year, the function makes one recursive call, reducing the value of years by 1 in each step until it reaches the base case (when years == 0). Therefore, the number of recursive calls made is directly proportional to the value of years. The space complexity is also O(n) because each recursive call occupies a frame in the program’s call stack until the base case is reached, and only then does it start unwinding.

While this recursive solution works well for small values of years, it can lead to stack overflow errors if the input is very large because of the excessive number of recursive calls. Additionally, since there are no overlapping subproblems in this particular case (unlike Fibonacci or factorial recursion), memoization does not apply here.

A better approach for practical scenarios is to optimize the recursion by converting it into an iterative algorithm. The iterative version calculates the result by looping over the number of years, multiplying the future value by (1 + growthRate) in each iteration. This reduces the space complexity to O(1) by eliminating the need for the call stack while retaining the same O(n) time complexity. Thus, the iterative solution is more memory-efficient and avoids the risks associated with recursion when dealing with larger datasets or timeframes.

**OUTPUT:**

