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

**1. Understand Asymptotic Notation:**

**Big O Notation:**

Big O notation describes the **upper bound** of an algorithm’s **time or space complexity**, showing how it scales with input size (n).

How Big O helps in analysing algorithms:

* **Compare Efficiency:**  
  Focuses on the **dominant term**, ignoring constants → helps compare algorithms.
* **Find Bottlenecks:**  
  Reveals slow-growing parts of code for large inputs.
* **Choose the Right Algorithm:**  
  E.g., Binary Search (O(log n)) is better than Linear Search (O(n)) for sorted data.
* **Predict Performance:**  
  Helps estimate performance as input size increases.

| **Notation** | **Name** | **Example** | **Growth** |
| --- | --- | --- | --- |
| O(1) | Constant | Array index access | Fastest |
| O(log n) | Logarithmic | Binary search | Very efficient |
| O(n) | Linear | Linear search | Grows steadily |
| O(n log n) | Linearithmic | Merge sort, quick sort | Efficient sort |
| O(n²) | Quadratic | Bubble sort, nested loops | Slower |
| O(2ⁿ) | Exponential | Recursive Fibonacci | Very slow |

* **Best for small inputs:** O(n²) and higher
* **Best for large inputs:** O(log n), O(n), or O(n log n).
* Always **prefer lower time complexity** when input size is large.

**Best, Average, and Worst-Case:**

Best Case:

* **Linear Search:** Target is the **first element**.
* **Binary Search:** Target is at the **middle index** initially.

Average Case:

* **Linear Search:** Target is **somewhere in the middle**.
* **Binary Search:** **Each step halves** the search space → log₂(n) comparisons.

Worst Case:

* **Linear Search:** Target is at the **end** or **missing**.
* **Binary Search:** Element is **not found** after all splits.

| **Big O** | **Name** | **What it Means** |
| --- | --- | --- |
| **O(1)** | Constant Time | Same number of steps regardless of input size |
| **O(n)** | Linear Time | Steps grow **directly with input size** |
| **O(log n)** | Logarithmic Time | Steps grow **slowly** even as input size increases |
|  |  |  |

* **Binary Search is more efficient** than Linear Search on **sorted data**.
* Choose the **search strategy** based on:
  + **Input size**
  + **Data order**
  + **Performance needs**

**2. Setup and Implementation:**

**Code:**

Product.java:

public class Product {

private int productId;

private String productName;

private String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

@Override

public String toString() {

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

}

}

SearchService.java:

import java.util.Arrays;

import java.util.Comparator;

public class SearchService {

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

for (Product product : products) {

if (product.getProductId() == targetId) {

return product;

}

}

return null;

}

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int midId = products[mid].getProductId();

if (midId == targetId) {

return products[mid];

} else if (midId < targetId) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void sortProductsById(Product[] products) {

Arrays.sort(products, Comparator.comparingInt(Product::getProductId));

}

}

Main.java:

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Watch", "Accessories"),

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

};

int targetId = 103;

Product foundLinear = SearchService.linearSearch(products, targetId);

System.out.println("Linear Search Result: " + (foundLinear != null ? foundLinear : "Not found"));

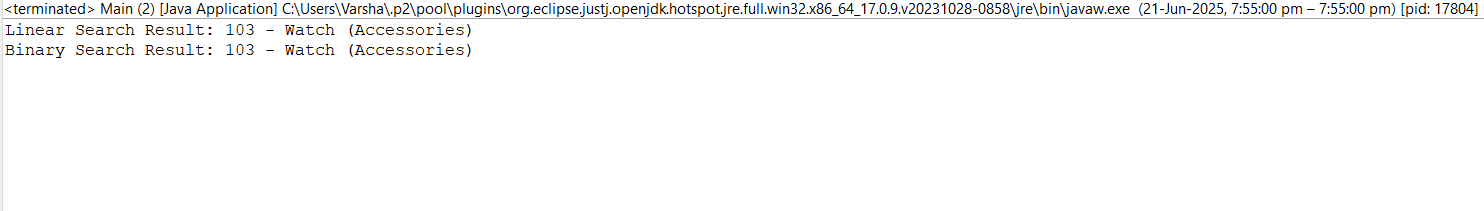
SearchService.sortProductsById(products);

Product foundBinary = SearchService.binarySearch(products, targetId);

System.out.println("Binary Search Result: " + (foundBinary != null ? foundBinary : "Not found"));

}

}

**Output:**

**3. Analysis:**

**Time Complexity Analysis:**

Linear search has a time complexity of O(n), while binary search has a time complexity of O(log n). This means that binary search is significantly faster than linear search, especially for large datasets, because it divides the search space in half with each step. However, binary search requires the data to be sorted, whereas linear search does not.

Linear Search:

* **Time Complexity:** O(n) in the worst and average cases, O(1) in the best case.
* **How it works:** It sequentially checks each element in the list until the target element is found or the end of the list is reached.
* **Use Cases:** Suitable for small datasets or when the data is not sorted.

Binary Search:

* **Time Complexity:** O(log n) in all cases (best, average, and worst).
* **How it works:** It repeatedly divides the search interval in half. It compares the middle element of the interval with the target value. If they match, the search is successful. If the target is smaller, the search continues in the left half; otherwise, it continues in the right half.
* **Use Cases:** Efficient for large, sorted datasets.
* **Requirement:** The data must be sorted.

Comparison:

| **Algorithm** | **Time Complexity** | **Best Use Case** |
| --- | --- | --- |
| **Linear Search** | O(n) | Small datasets, unsorted data |
| **Binary Search** | O(log n) | Large, sorted datasets |

Key Benefits:

* **Increased Engagement:**  
  Analyses likes, watch time, shares, etc., to show content users are likely to enjoy.
* **Improved User Experience:**  
  Curated, relevant content keeps users active and returning.
* **Better Content Discovery:**  
  Helps users find new content and creators beyond what’s trending or recent.
* **Adaptability:**  
  Learns and evolves with user behaviour to stay relevant.

**Most Suitable Algorithm: Personalized Recommendation Algorithm**

Why It's Suitable:

1. **User-Centric Experience:**

* Delivers content based on individual preferences, boosting relevance and satisfaction.

1. **Higher Engagement:**

* Users are more likely to interact (like, comment, share) with content tailored to their interests.

1. **Dynamic and Adaptive:**

* Learns from user behaviour over time and adapts to changing preferences.

1. **Improved Content Discovery:**

* Surfaces new and diverse content users may not find through manual browsing.

Why Other Algorithms Are Less Suitable:

* **Generic Algorithms** (e.g., sorting by time or popularity):
  + Don’t account for personal taste → leads to a bland or irrelevant feed.
* **Task-Specific Algorithms** (e.g., image recognition, NLP):
* Supportive for moderation or tagging, but not for personalized content delivery.

**Conclusion:**

A personalized recommendation algorithm is the most suitable choice for a social media platform because it enhances engagement, improves user satisfaction, and ensures the feed remains relevant and dynamic for each user.

**Exercise 7: Financial Forecasting**

**1. Understand Recursive Algorithms:**

**Recursion** is a programming technique where a function **calls itself** to solve a problem by breaking it down into smaller subproblems of the same type.

A recursive function typically has:

1. **Base Case** – The simplest case that stops the recursion.
2. **Recursive Case** – The part where the function calls itself with a smaller input.

Recursion can **simplify code** for problems that are:

* Naturally repetitive or self-similar
* Involve hierarchical structures (e.g., trees, graphs)
* Require exploration of multiple paths (e.g., backtracking)

Examples of Problems Simplified by Recursion:

| **Problem** | **Why Recursion Helps** |
| --- | --- |
| **Factorial (n!)** | Each factorial can be broken into smaller ones |
| **Fibonacci Series** | Depends on values of previous terms |
| **Tree Traversals** | Naturally recursive due to node hierarchy |
| **Directory Traversal** | Folders contain folders (nested structure) |
| **Maze Solving** | Explore paths recursively with backtracking |

Be careful with recursion:

* Ensure a **base case** to avoid infinite loops.
* Deep recursion can cause a **stack overflow**.
* May need optimization (e.g., **memoization** or converting to **iteration**) for performance.

Recursion is powerful for solving **complex problems with simple logic**, especially when the problem can be **broken into smaller, similar subproblems**.

**2. Setup and Implementation:**

The basic compound interest formula:

Future Value (FV)=Present Value (PV)×(1+r)n

Where:

* PV = present value (initial amount)
* r = growth rate (e.g., 0.05 for 5%)
* n = number of years

**Code:**

class Main {

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

double presentValue = 10000; // Starting amount

double growthRate = 0.08; // 8% growth rate

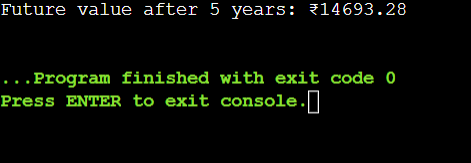
int years = 5;

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

System.out.printf("Future value after %d years: ₹%.2f%n", years, futureValue);

}

**Output:**

****

**3. Analysis:**

Analyzing Time Complexity of Recursive Algorithms:

**1. Identify Recursive Behaviour:**

* **Recursive Calls:** How many times does the function call itself?
* **Input Reduction:** How does the input shrink each time? (e.g., n → n-1, n/2)

**2. Define the Recurrence Relation:**

* Express time as: T(n) = ...
* Example:
  + If a function halves input and calls itself twice:  
    T(n) = 2 \* T(n/2) + O(1)

**3. Solve the Recurrence:**

Use one of the following methods:

* **Recursion Tree** – Visualizes work per level.
* **Master Theorem** – Directly solves divide-and-conquer relations.
* **Substitution** – Guess & prove by induction.

**Example: Fibonacci**

def fibonacci(n):

if n <= 1:

return n

return fibonacci(n-1) + fibonacci(n-2)

Other Considerations:

| **Concept** | **Description** |
| --- | --- |
| **Base Case** | Stops recursion; usually O(1) time |
| **Space Complexity** | Stack space grows with depth (can be O(n)) |
| **Tail Recursion** | Optimized in some languages to save space |

**Techniques to Optimize Recursion:**

**Memoization:**

* **What it does:** Caches results of function calls.
* **When to use:** Recursive problems with **overlapping subproblems** (e.g., Fibonacci).
* **Benefit:** Avoids redundant calculations.

Example:

def fibonacci(n, memo={}):

if n in memo:

return memo[n]

if n <= 2:

return 1

memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)

return memo[n]

**Dynamic Programming (DP):**

* **What it does:** Solves subproblems **bottom-up** using a table.
* **When to use:** Optimization problems with **optimal substructure** (e.g., knapsack, shortest path).
* **Benefit:** Systematic and fast; avoids recursion overhead.

**Iterative Solutions:**

* **What it does:** Uses loops instead of recursion.
* **When to use:** To prevent **stack overflow** or improve efficiency.
* **Benefit:** More memory-efficient; often faster for large inputs.

**Well-defined Base and Recursive Cases:**

* **Base Case:** Stops recursion (e.g., if n == 0 return 1)
* **Recursive Step:** Moves toward base case (e.g., n-1 or n/2)
* **Importance:** Prevents infinite loops and excessive computation.

**Compiler Optimizations:**

* **Tail Recursion:** Last call in function; can be optimized by compiler.
* **Compiler Flags:** Enable with -O2, -O3 (in C/C++ compilers) for speed.

**Summary Table:**

| **Technique** | **Purpose** | **Benefit** |
| --- | --- | --- |
| Memoization | Cache repeated calls | Reduces redundancy |
| Dynamic Programming | Bottom-up solving | Improves speed & structure |
| Iterative Conversion | Replace recursion | Saves memory & avoids stack |
| Base Case Check | Stop infinite recursion | Ensures correctness |
| Tail Recursion | Optimize call stack | Better performance (if supported) |