**Ex:1 Inventory Management System**

**1.Why are Data Structures and Algorithms Essential?**

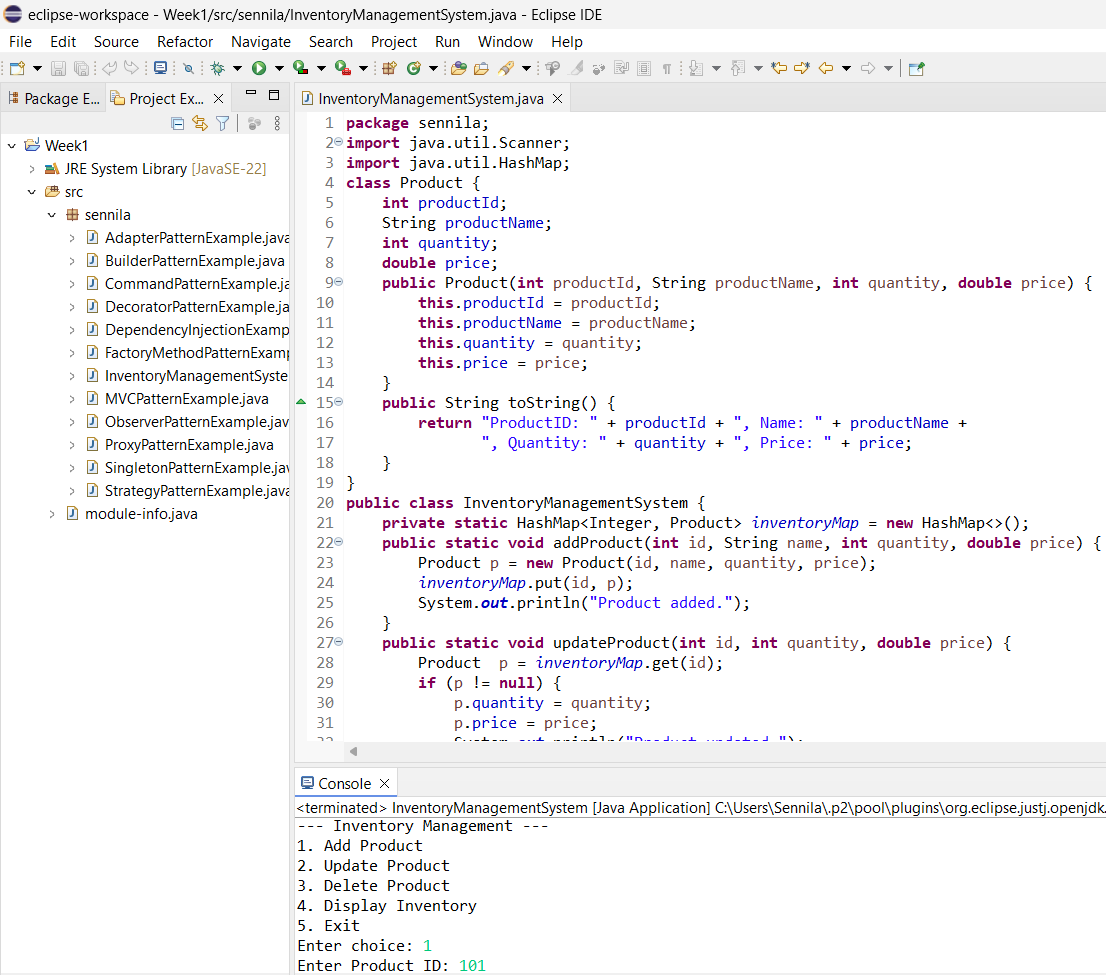
* Scalability: Large inventories can contain thousands of items. Efficient data structures ensure smooth handling and quick access.
* Performance: Operations like searching for a product, updating stock, or deleting discontinued items must be fast to avoid performance bottlenecks.
* Memory Management: Proper data structures optimize memory usage and avoid overhead.

**Suitable Data Structures:**

* **HashMap:**
  + Key: productId
  + Value: Product object
  + Fast lookup, insertion, deletion in O(1) average time.
* **ArrayList:**
  + Ordered list of products.
  + Slower for searching (O(n)) or deletion.
* **TreeMap:**
  + Sorted by key (productId)
  + If sorted access is needed. Operations are O(log n).

Best Choice: HashMap<Integer, Product> for optimal performance in real-time lookup and updates.

**2,3-Setup and Implementation**



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.**4.Analysis:**

**Time Complexity (using HashMap):**

| Operation | Time Complexity |
| --- | --- |
| Add Product | O(1) average |
| Update Product | O(1) average |
| Delete Product | O(1) average |
| Display Inventory | O(n) (where n is the number of products) |
| **OPTIMISATION IDEAS:** |  |
| (1) Use TreeMap for sorted/range queries,  (2) Use ConcurrentHashMap for thread-safe access.  (3) Minimize memory with immutable objects,  (4) Use DB (SQLite/MongoDB) for persistent storage. |  |
|  |  |

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

**1.Understand Asymptotic Notation**

**🔹 Big O Notation**

* Big O notation describes the upper bound on the time or space complexity of an algorithm.
* It helps in analyzing how an algorithm performs as the input size n grows.

**🔹 Best, Average, and Worst Cases in Search**

| **Scenario** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Best Case | O(1) – first match | O(1) – middle match |
| Average Case | O(n/2) ≈ O(n) | O(log n) |
| Worst Case | O(n) – last match/miss | O(log n) – item not found |
|  |  |  |
|  |  |  |
|  |  |  |

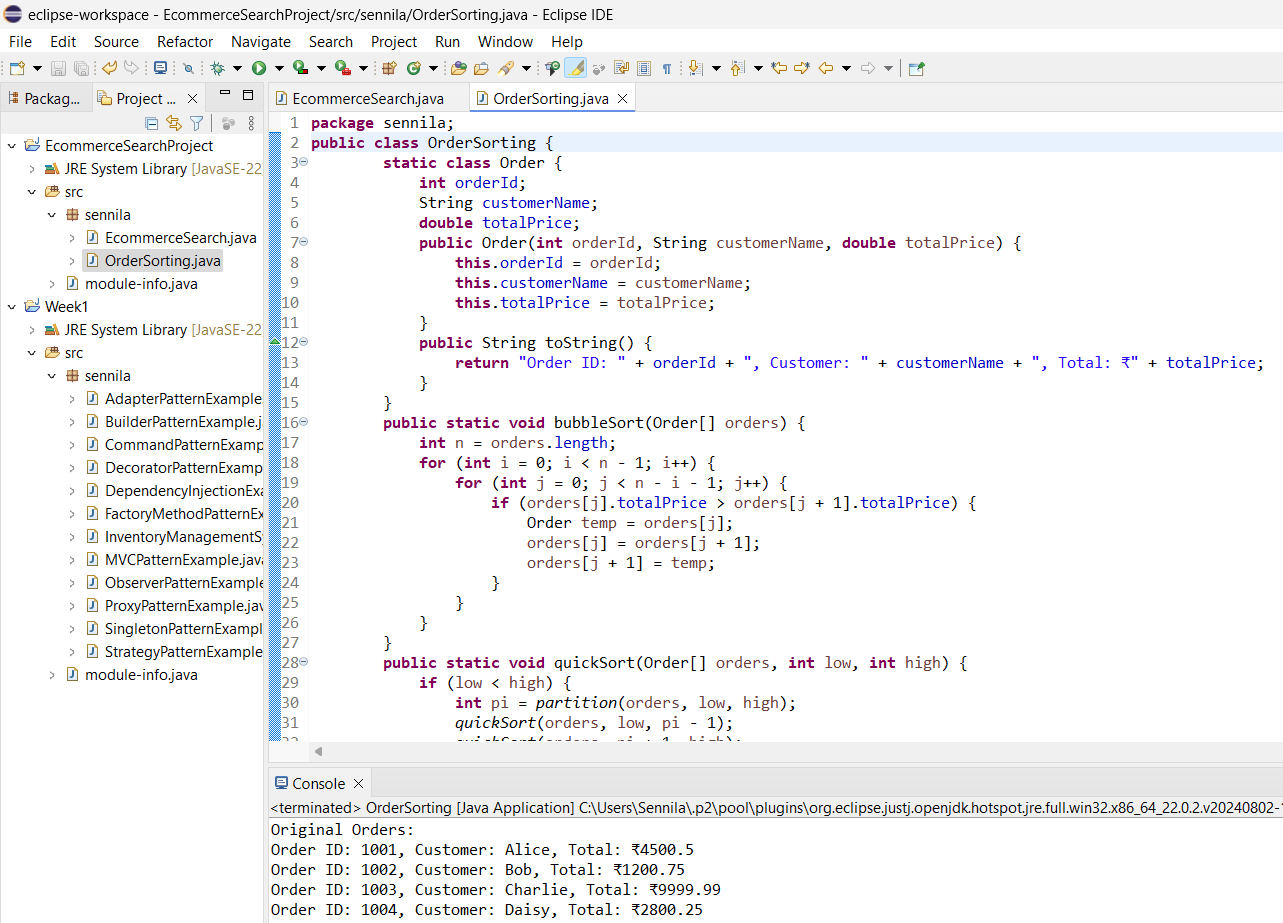
**2,3-Setup and Implementation**

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

****

**4.Analysis & Comparison**

| Algorithm | Time Complexity | Suitable When |
| --- | --- | --- |
| Linear | O(n) | Small or unsorted product lists |
| Binary | O(log n) | Large, sorted product lists |

**Binary Search is more suitable** for an e-commerce platform because:

1. **Faster Performance**:
   * With O(log n) complexity, binary search drastically reduces lookup time for large product catalogs (e.g., millions of items).
2. **Scalability**:
   * E-commerce platforms scale rapidly. Binary search ensures that performance remains efficient as inventory grows.
3. **Sorted Data is Common**:
   * Products are often sorted (by name, price, popularity), so maintaining a sorted structure (like a database index or a sorted array) aligns well with binary search.

**When Linear Search Might Be Useful:**

* For small datasets
* If products are unsorted or dynamically changing frequently
* When doing partial/regex-based search, which binary search can’t handle

**Ex:3 Sorting Customer Orders**

**1.Understanding Sorting Algorithms**

**🔹 Bubble Sort:**

* Repeatedly swaps adjacent elements if they are in the wrong order.
* Time Complexity: O(n²) worst case.
* Simple but inefficient for large datasets.

**🔹 Insertion Sort:**

* Builds the sorted list one element at a time.
* Time Complexity: O(n²) worst case, O(n) best case (nearly sorted).
* Good for small or nearly sorted arrays.

**🔹 Quick Sort:**

* Uses divide-and-conquer. Picks a pivot, partitions the array, and recursively sorts.
* Time Complexity: O(n log n) average, O(n²) worst case (rare).
* Fast and efficient for large datasets.

**🔹 Merge Sort:**

* Divides the array, sorts both halves, and merges them.
* Time Complexity: O(n log n) always.
* Stable and predictable but uses more memory.

**2,3-Setup and Implementation**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**4.Analysis**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space** |
| --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) (rare) | O(log n) |
|  |  |  |  |  |

**Why Quick Sort is Preferred:**

* Much faster than Bubble Sort for large datasets.
* Lower time complexity on average.
* Optimized for in-place sorting and better suited for performance-critical applications like e-commerce platforms.

**EX:4 Employee Management System**

**1.Understand Array Representation**

**How Arrays Are Represented in Memory:**

* **Contiguous Memory Block:**Arrays are stored in a single continuous block of memory. This means each element is placed right next to the previous one in memory.
* **Fixed Size at Declaration:**When you create an array, Java allocates a block of memory big enough to hold all the elements you specified. You cannot resize the array after creation.
* **Index-Based Access:**Each element in an array can be directly accessed by its index. Internally, this uses pointer arithmetic to jump directly to the memory address:

Address of arr[i] = base\_address + (i × size\_of\_element)

**Advantages of Arrays:**

1. **Fast Element Access (O(1) Time):**Accessing any element using its index is done in constant time. This makes searching by index or traversing highly efficient.
2. **Cache-Friendly:**Because array elements are stored contiguously, CPUs can fetch data into cache lines more efficiently, speeding up read operations.
3. **Simple and Easy to Use:**Arrays are a basic, well-understood data structure that’s simple to declare and use in code.
4. **No Overhead of Additional Pointers:**Unlike linked lists, arrays don’t store extra references (like next/prev), which saves memory.

**2,3-Setup and Implementation**

**A screenshot of a computer

AI-generated content may be incorrect.**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

**4.Time Complexity of Operations Using Arrays**

| **Operation** | **Description** | **Time Complexity** |
| --- | --- | --- |
| Add | Insert at end (if space is available) | O(1) |
|  | Insert at specific index | O(n) (due to shift) |
| Search | Find by employeeId (linear search) | O(n) |
| Traverse | Print or visit each employee | O(n) |
| Delete | Remove by shifting elements | O(n) |

**Explanation:**

* Add at end: Direct assignment is fast if there is space (employees[count++] = emp).
* Add at middle: Requires shifting elements to the right → O(n).
* Search: Without sorting or indexing, search requires checking each item.
* Traverse: One loop over all items.
* Delete: After finding the element, you must shift all subsequent elements to fill the gap.

**Limitations of Arrays**

1. **Fixed Size:**Once declared, you cannot increase or decrease the array size dynamically. You must know the maximum number of elements in advance.
2. **Wasted or Insufficient Memory:**
   * Allocate too much: unused memory is wasted.
   * Allocate too little: you can’t add more elements later.
3. **Insertion/Deletion is Expensive:**Adding or removing items at any position other than the end requires shifting elements, which is inefficient.
4. **No Built-in Methods for Flexibility:**Unlike ArrayList, arrays lack methods like .add() or .remove(). Manual code is needed for every action.

**When to Use Arrays**

Use arrays when:

* The number of elements is fixed or known in advance.
* You need fast index-based access.
* Memory usage needs to be tightly controlled.
* You are working on low-level logic or performance-critical code.

**Ex:5 Task Management System**

**1.Understand Linked Lists**

**What is a Linked List?**

A linked list is a linear data structure where elements (called nodes) are stored in non-contiguous memory locations, and each node points to the next (or previous) node using pointers.

**🔸 Types of Linked Lists**

**1. Singly Linked List**

* Each node contains:
  + Data (task info, etc.)
  + Next pointer pointing to the next node
* The last node points to null
* Can only be traversed in one direction

**Structure Example:**

[Task1 | next] → [Task2 | next] → [Task3 | null]

**Pros:**

* Simple and uses less memory per node
* Efficient insertion/deletion at beginning

**Cons:**

* Cannot traverse backward
* Deletion of a specific node requires previous node's reference

**2. Doubly Linked List**

* Each node contains:
  + Data
  + Next pointer
  + Previous pointer
* Can be traversed forward and backward

**Structure Example:**

null ← [Task1 | prev, next] ↔ [Task2 | prev, next] ↔ [Task3 | prev, null]

**Pros:**

* Bidirectional traversal
* Easy to delete any node (even without previous reference)

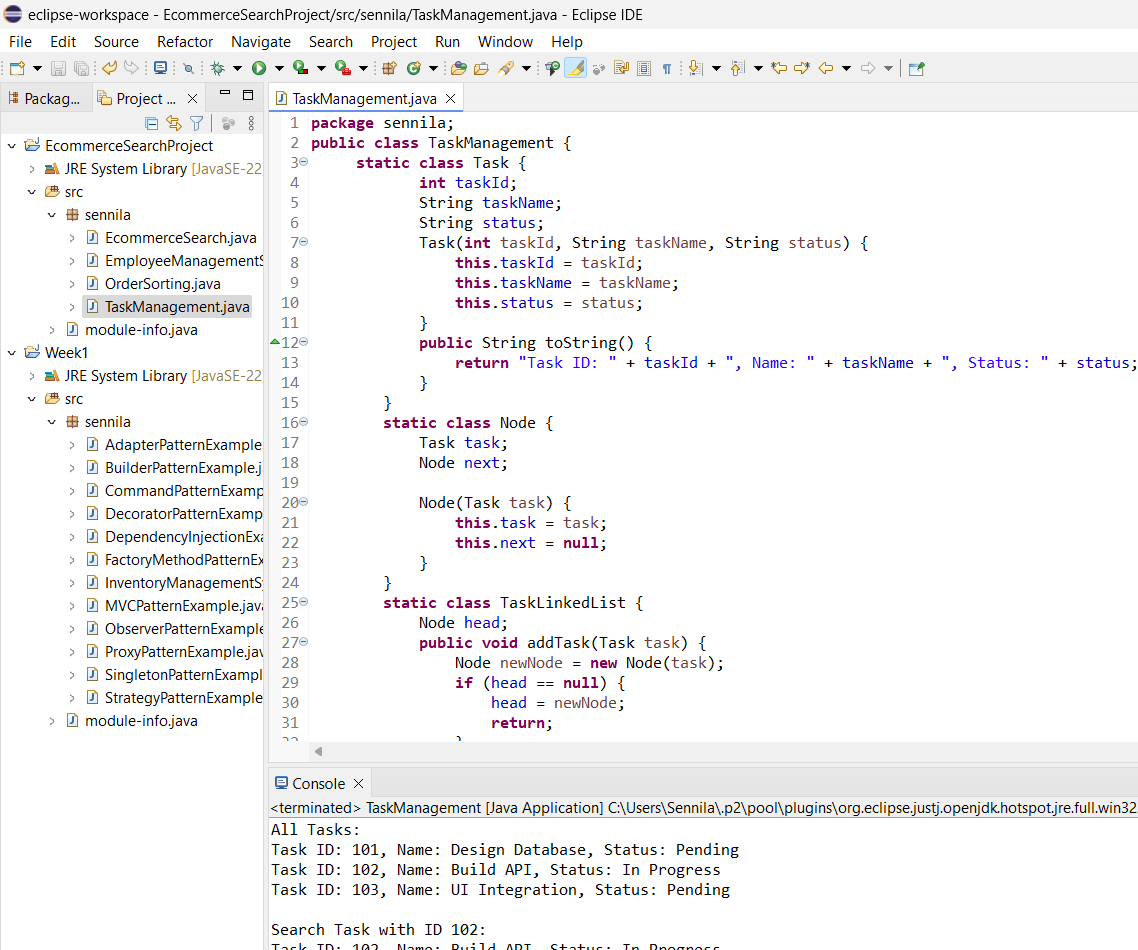
**Cons:**

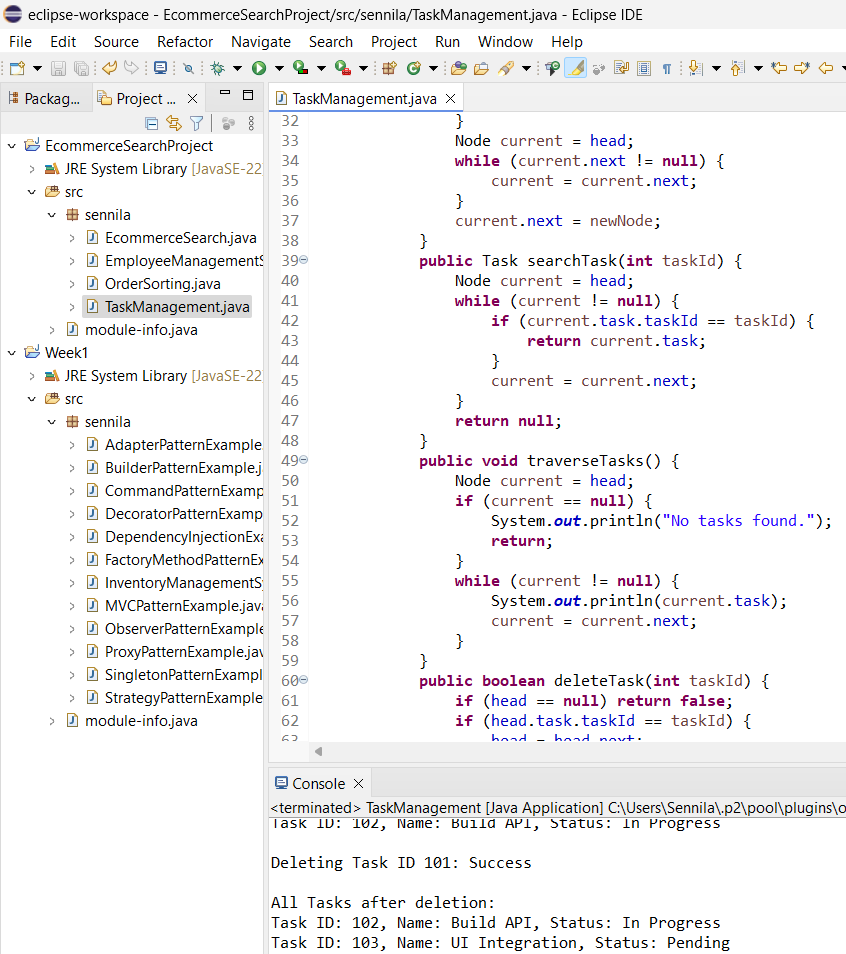
* Requires more memory (extra pointer)
* Slightly more complex to manage

**🔍 When to Use Each**

| **Use Case** | **Choose** |
| --- | --- |
| Efficient memory use, simple operations | Singly Linked List |
| Need for backward traversal or fast deletion anywhere | Doubly Linked List |

**2,3-Setup and Implementation**





A screenshot of a computer

AI-generated content may be incorrect.

**4.Time Complexity Analysis:**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| addTask | O(n) | Traverses list to add at the end. |
| searchTask | O(n) | Checks each node until found or end reached. |
| traverseTasks | O(n) | Visits all nodes to print tasks. |
| deleteTask | O(n) | Searches for task by ID and deletes if found. |

**Advantages of Linked Lists over Arrays:**

* **Dynamic Size:** No need to define size at the beginning; memory is allocated as needed**.**
* **Efficient Insert/Delete:** Insertions and deletions are faster (O(1) at head, O(n) elsewhere) since no shifting is required.
* **No Wasted Space:** Only the needed memory is used, unlike arrays where unused elements occupy space.
* **Useful for Unknown or Varying Data Size:** Ideal when the number of tasks is unknown or frequently changing.

**Ex:6 Library Management System**

**1.Linear Search**

Linear search is a simple search algorithm where you check each element one by one until you find a match or reach the end of the list.

**📚 Use Case in Library:**

* If your books are stored unsorted (e.g., in the order they were added), linear search is the only option.
* For example, to find a book by title, you start from the first book and compare each title with the target title.

**Binary Search**

Binary search is a fast search algorithm that works on a **sorted** array or list. It repeatedly divides the search space in half.

**📚 Use Case in Library:**

* If books are sorted alphabetically by title or author, binary search is much faster.
* For example, to find a book titled *"Gone Girl"*, you start at the middle, then narrow down the range depending on alphabetical order.

**2,3-Setup and Implementation**

A screenshot of a computer

AI-generated content may be incorrect.

**A screenshot of a computer

AI-generated content may be incorrect.**

**4.Time Complexity Comparison**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** | **Requires Sorted Data?** |
| --- | --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) | ❌ No |
| Binary Search | O(1) | O(log n) | O(log n) | ✅ Yes |

* n = number of books in the list

**When to Use Each Algorithm**

**🔹 Linear Search**

* **Use when:**
  + Data is unsorted
  + The list is small or medium-sized
  + Simplicity is more important than speed
* **Examples:**
  + Searching in a temporary or unsorted list
  + Book inventory loaded without sorting

**🔹 Binary Search**

* **Use when:**
  + Data is sorted alphabetically by title or author
  + The list is large, and performance matters
* **Examples:**
  + Searching in a large digital library
  + Searching in a pre-sorted database or catalog

**Ex:7 Financial Forecasting**

**1.Understanding Recursive Algorithms**

**What is Recursion?**

Recursion is a programming technique where a function calls itself to solve a problem in smaller, simpler subproblems. Each recursive call moves the problem closer to a base case (a condition that stops the recursion).

**Structure of a Recursive Function**

returnType function(parameters) {

if (base case condition) {

return base case value;

} else {

return function(smaller problem); // recursive call

}

}

**Example (Mathematical):** Factorial

int factorial(int n) {

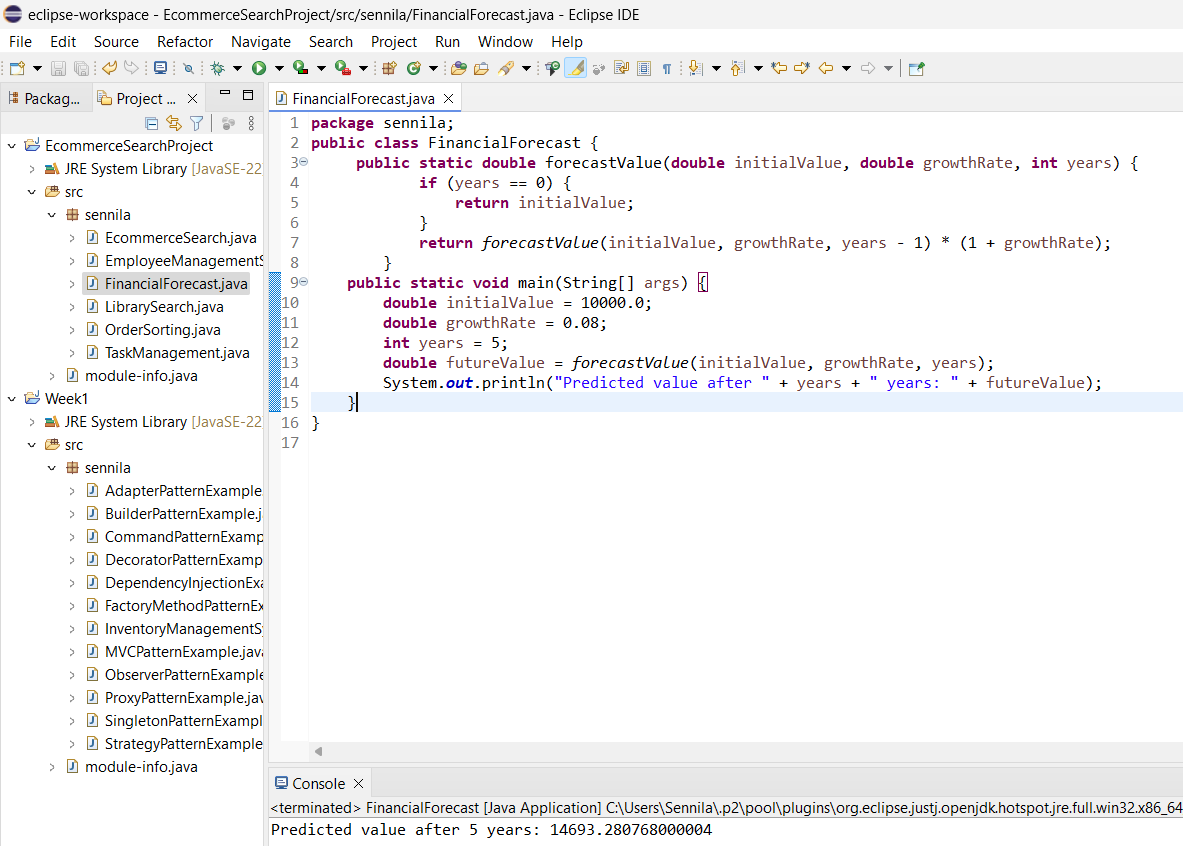
if (n == 0) return 1;

return n \* factorial(n - 1);

}

factorial(5) calls factorial(4), then factorial(3), and so on, until factorial(0).

**2,3-Setup and Implementation**



**4.Time Complexity**

**Recursive Function:**

forecastValue(initialValue, growthRate, years)

**📈 Time Complexity:**

* The recursive function makes one recursive call per year.
* So, total recursive calls = years
* **Time Complexity:**

O(n)

where n = number of years

**Optimization Techniques**

**1. Memoization (Top-Down Dynamic Programming)**

* Store already computed results in a Map or array to avoid recalculating them.
* Reduces redundant recursive calls in problems with overlapping subproblems.

**2. Convert to Iterative Approach**

* Use a **loop** to accumulate the value over time.
* Removes function call overhead and stack usage.
* Ideal for large datasets or limited system memory.