**DSA**

**MANDATORY:**

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

ASYMPTOTIC NOTATION

Big O Notation:

-Big O notation describes the upper bound of the time or space complexity of an algorithm in the worst-case scenario as the input size grows.

|  |  |
| --- | --- |
| Complexity | Description |
| O(1) | Constant time |
| O(log n) | Logarithmic time |
| O(n) | Linear time |
| O(n log n) | Linearithmic time |
| O(n²) | Quadratic time |

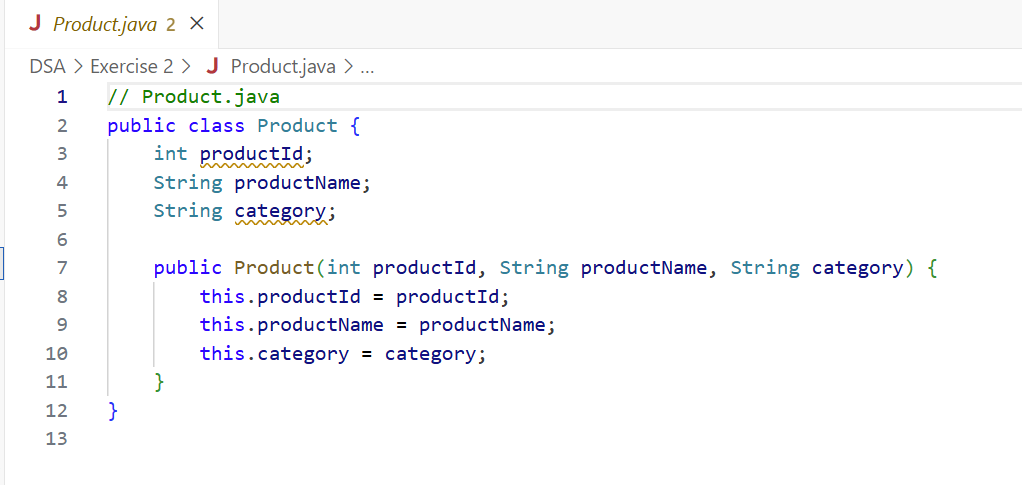
Linear Search: Binary Search (on sorted data):

Best: O(1) → if the element is the first Best: O(1)

Average: O(n/2) ≈ O(n) Average: O(log n)

Worst: O(n) Worst: O(log n)

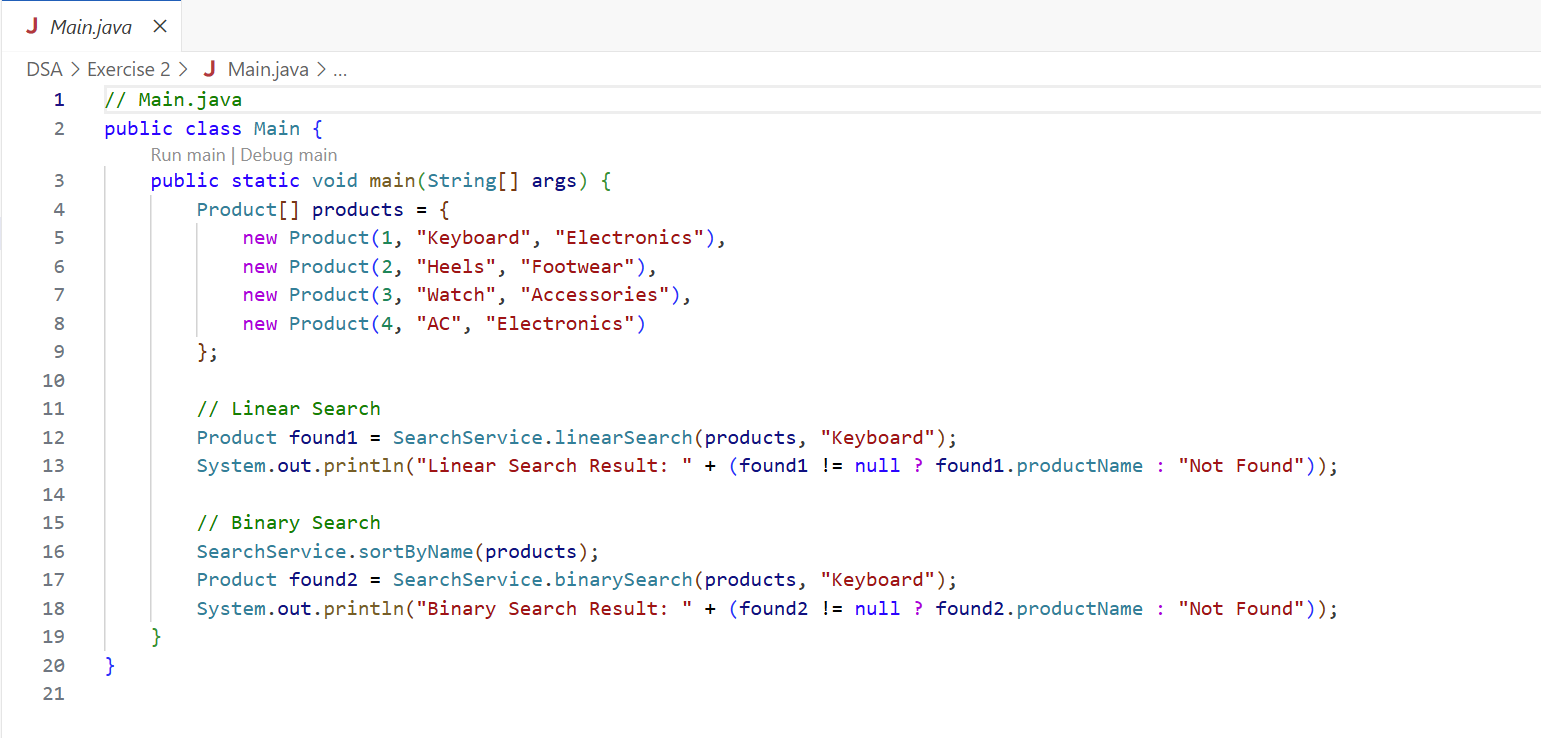
**Product.java:**



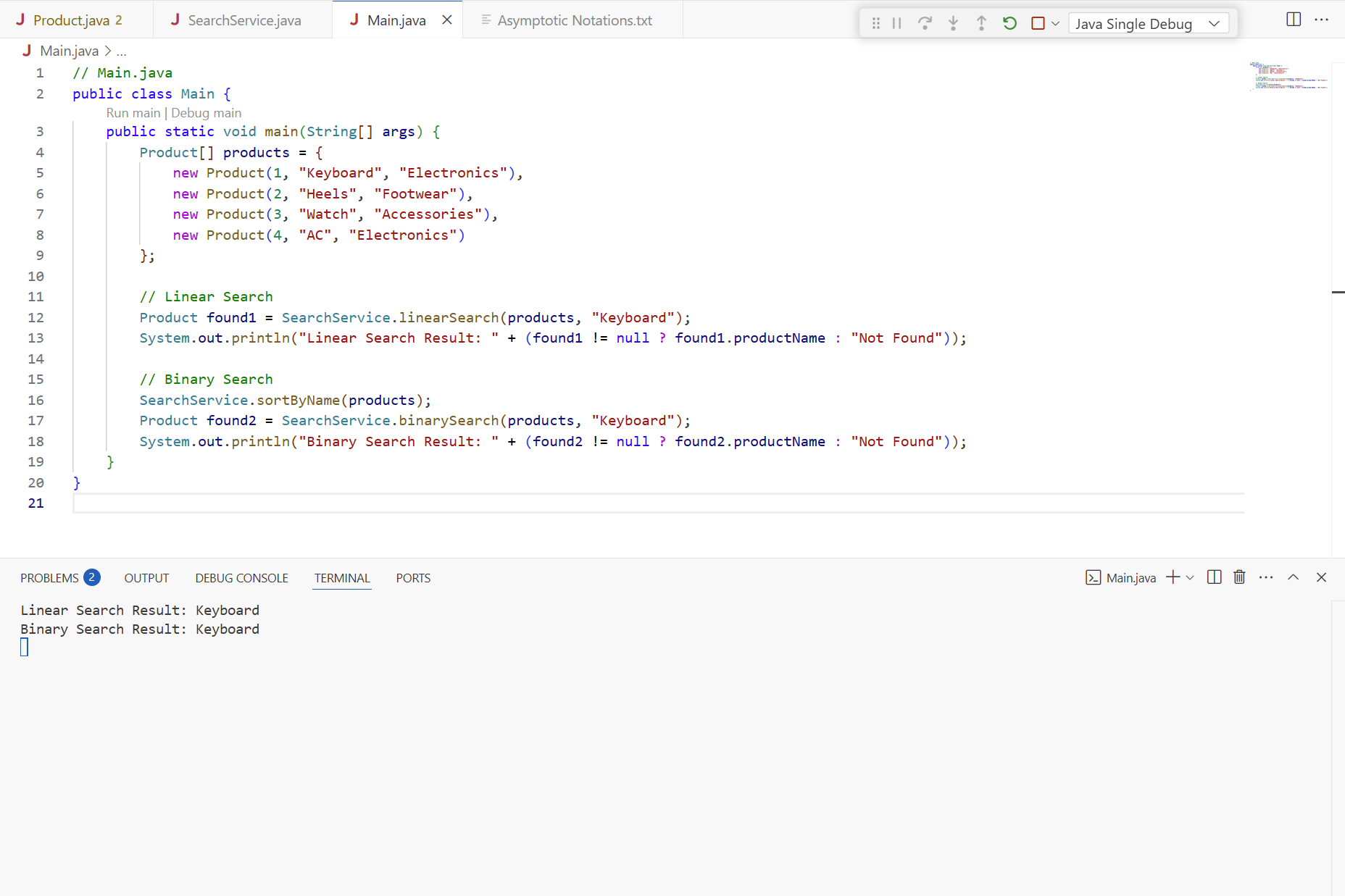
**SearchService.java:**



**Main.java:**



**OUTPUT:**

****

Analysis:

> Time Complexity:

|  |  |  |
| --- | --- | --- |
| Algorithm | Time Complexity | Suitable For |
| Linear Search | O(n) | Small or unsorted datasets |
| Binary Search | O(log n) | Large, sorted datasets |

> Which is more suitable for an E-commerce platform?

Binary Search is much faster than Linear Search for large datasets.

But it requires sorting beforehand (O(n log n)).

**Exercise 7: Financial Forecasting:**

What is Recursion?

- Recursion is a technique where a function calls itself to solve smaller instances of a problem.

Why Use Recursion?

- It breaks down complex problems into simpler subproblems.

- Often used for problems like factorial, Fibonacci, tree traversal, etc.

- In forecasting, it can naturally express repeated patterns like compound growth.

Setup:

We’ll assume the forecast follows compound growth:

Future Value 𝑛 = Future Value 𝑛-1 x (1 + Growth Rate)

We will:

Take initial value P

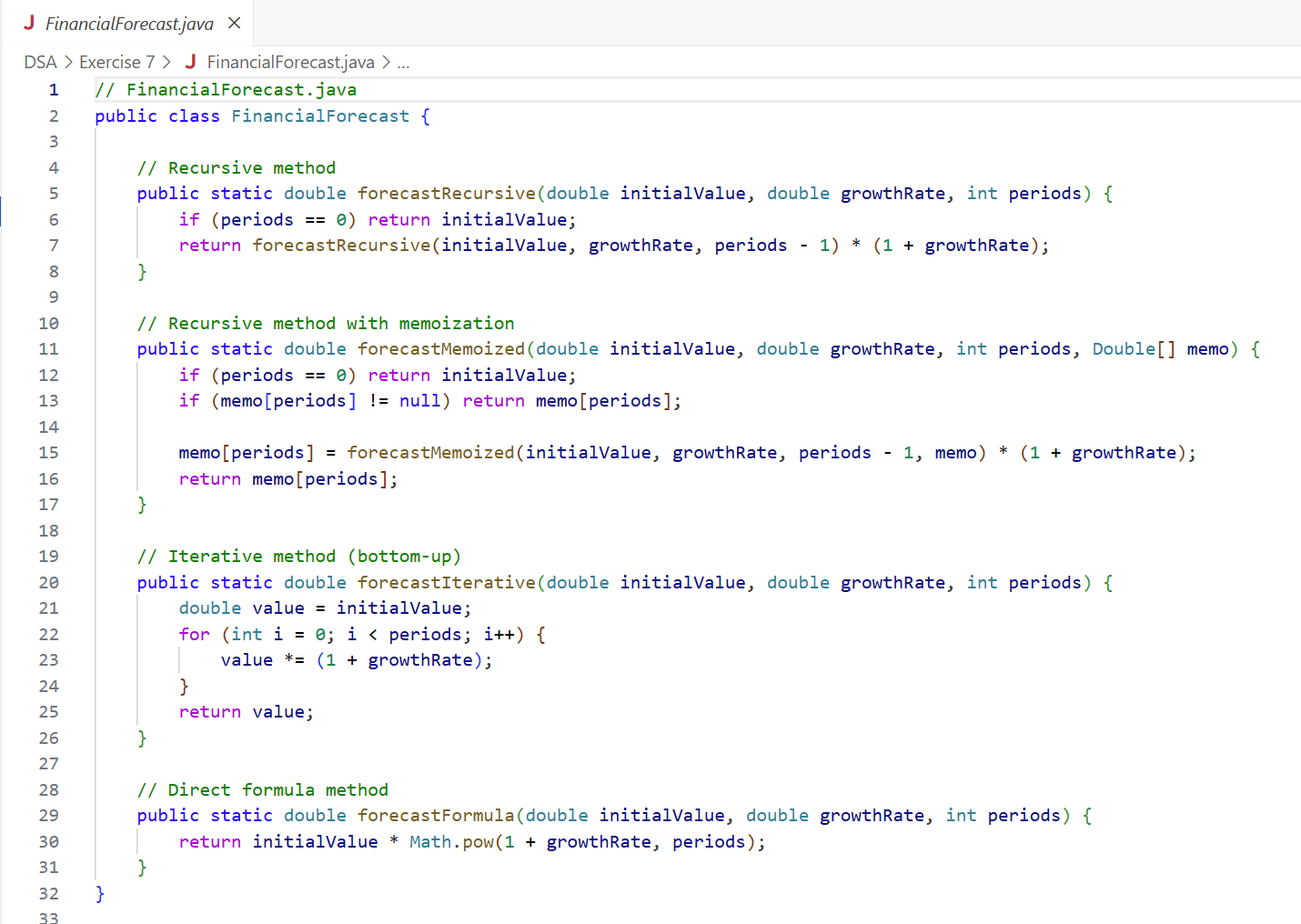
Growth rate r

Number of periods n

Compute value after n periods using recursion.

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Time Complexity | Space Complexity | Notes |
| Recursive | O(n) | O(n) | Simple but stack-heavy |
| Recursive + Memo | O(n) | O(n) | Avoids re computation |
| Iterative | O(n) | O(1) | More efficient in practice |
| Formula (Best) | O(1) | O(1) | Direct and most optimized |

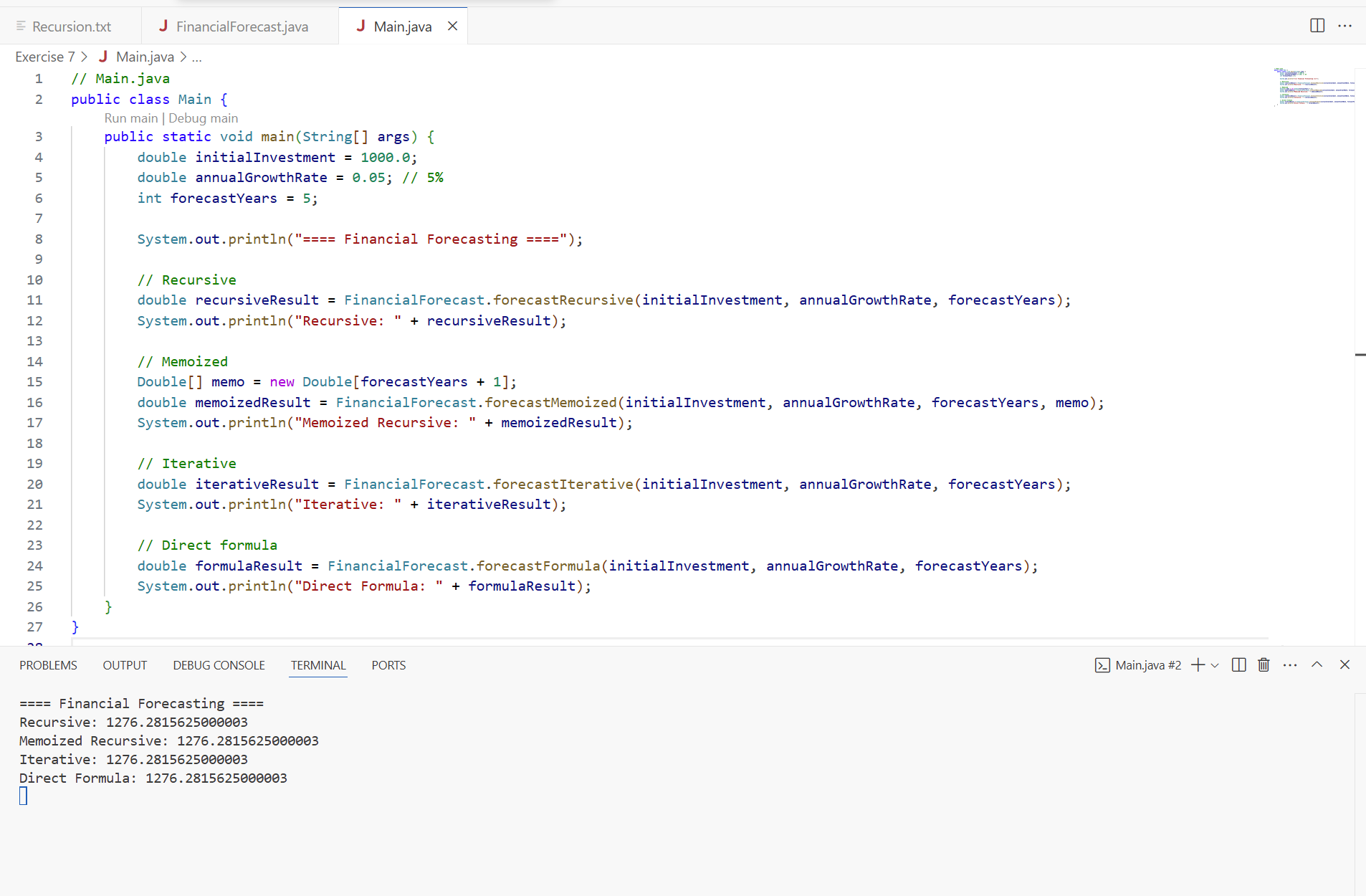
**FinancialForecast.java:**

****

**Main.java:**

****

**OUTPUT:**

****

**ADDITIONAL:**

**Exercise 1: Inventory Management System**

Why are data structures and algorithms essential in handling large inventories?

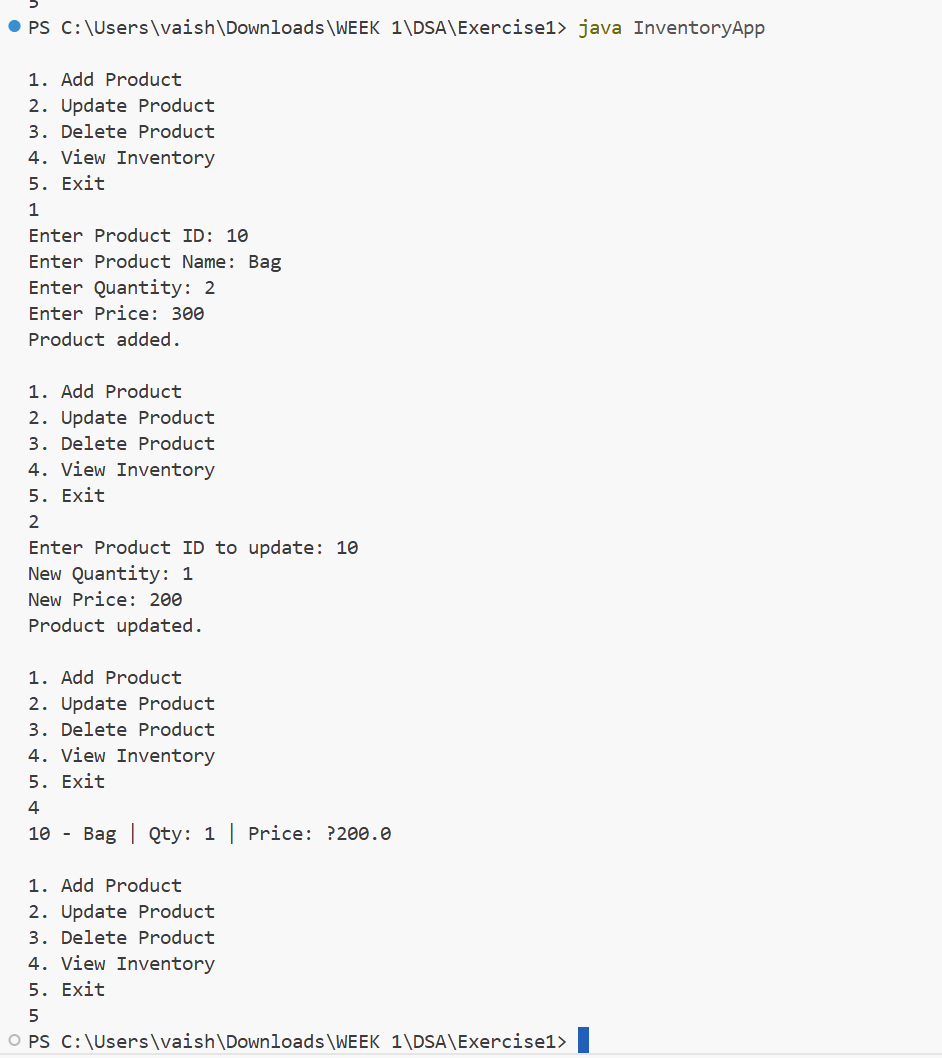
* Efficient Retrieval: In a warehouse, finding a product by its ID quickly is crucial. The right data structure allows constant or logarithmic time lookups.
* Efficient Updates: Inventory systems require frequent updates like restocking or sales. Algorithms ensure updates happen efficiently.
* Scalability: With thousands of products, a poor data structure can cause performance bottlenecks.
* Memory Optimization: Storing product details compactly and retrieving only necessary data saves memory and processing time.

Suitable Data Structures for Inventory Management:

| **Data Structure** | **Use Case** |
| --- | --- |
| ArrayList | Good for small datasets, sequential access |
| HashMap | Best for fast access by product ID (O(1) average time) |
| TreeMap | If data needs to be sorted by key |
| LinkedList | Rarely useful unless you have many insertions/deletions in the middle |
| Set | If duplicates must be avoided |

**Best choice for this case**: HashMap<String, Product> where productId is the key.

|  |  |
| --- | --- |
| **InventoryApp.java:** |  |
| **Product.java:** |  |
| **OUTPUT:** |  |



**Exercise 3: Sorting Customer Orders:**

**Understand Sorting Algorithms**

**🔹 Bubble Sort**

* Repeatedly steps through the list, compares adjacent elements, and swaps them if they’re in the wrong order.
* **Time Complexity:**
  + Best: O(n)
  + Average/Worst: O(n²)
* **Use Case:** Very small datasets or teaching purposes.

**🔹 Insertion Sort**

* Builds the final sorted array one item at a time by inserting elements into the correct position.
* **Time Complexity:**
  + Best: O(n)
  + Average/Worst: O(n²)

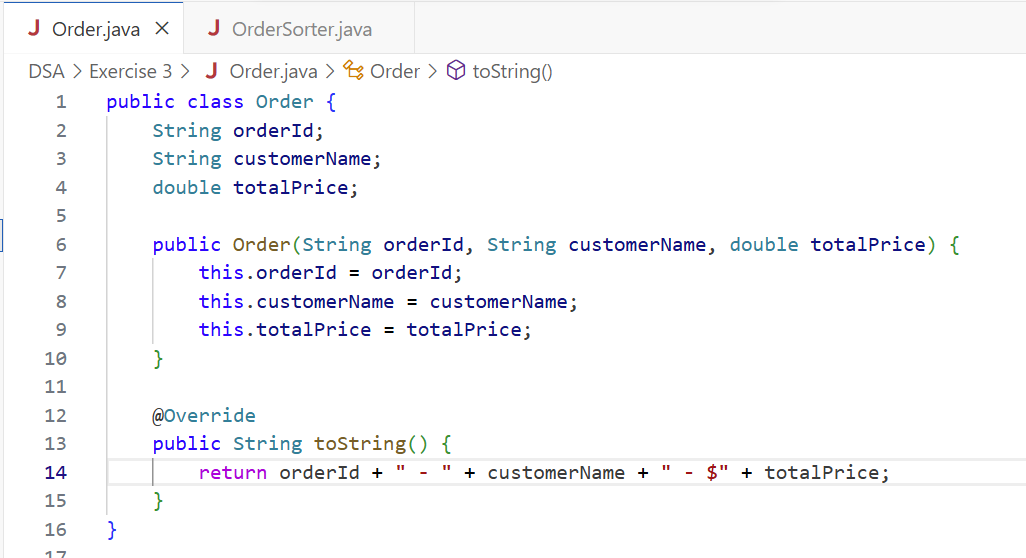
**🔹 Quick Sort**

* Divides the array using a **pivot**, recursively sorts the partitions.
* **Time Complexity:**
  + Best/Average: O(n log n)
  + Worst: O(n²) (but rare with good pivot choice)
* **Use Case:** General-purpose, highly efficient.

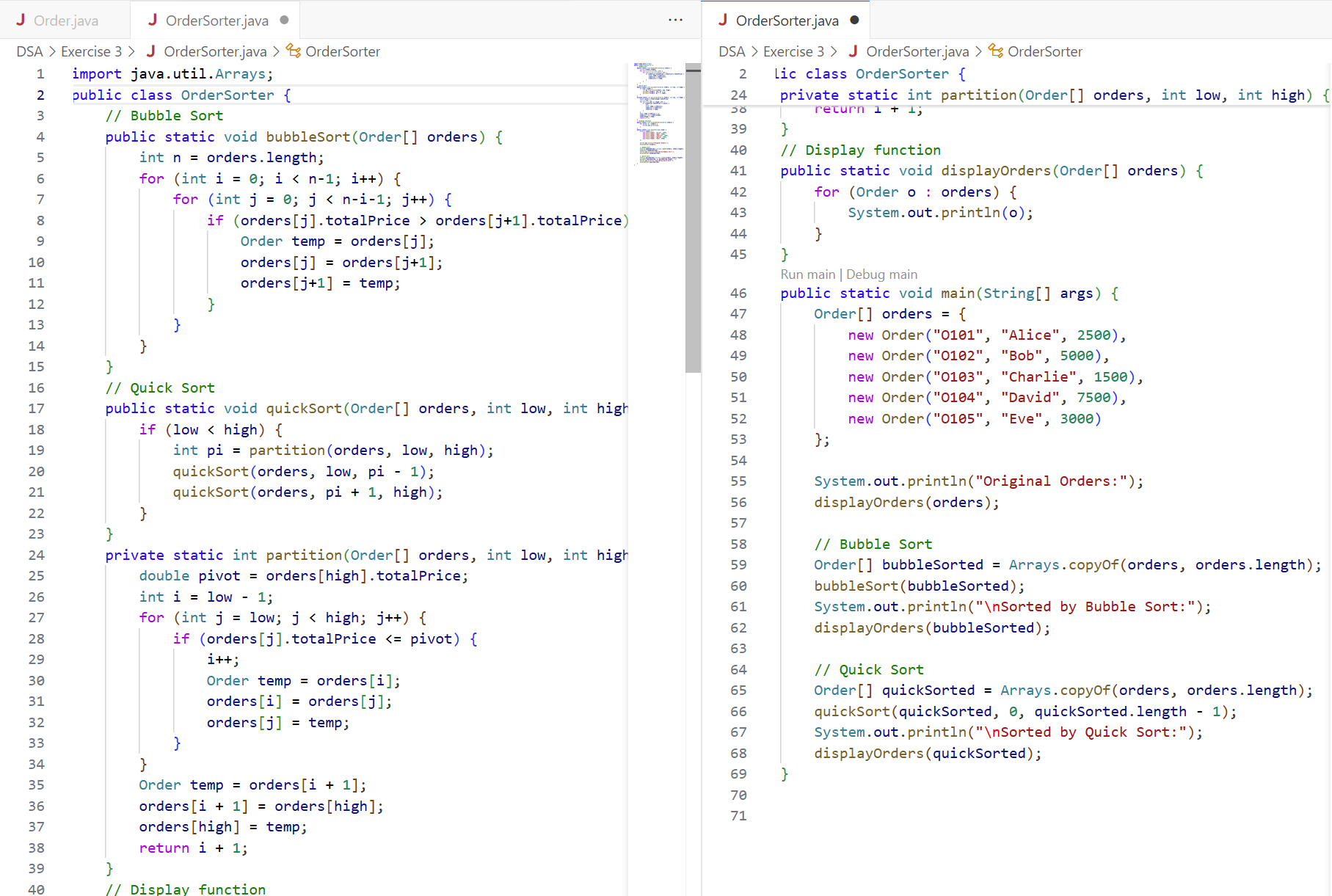
**🔹 Merge Sort**

* Divides the array into halves, recursively sorts and merges them.
* **Time Complexity:** O(n log n) in all cases
* **Use Case:** Stable sort, great for linked lists or large datasets.

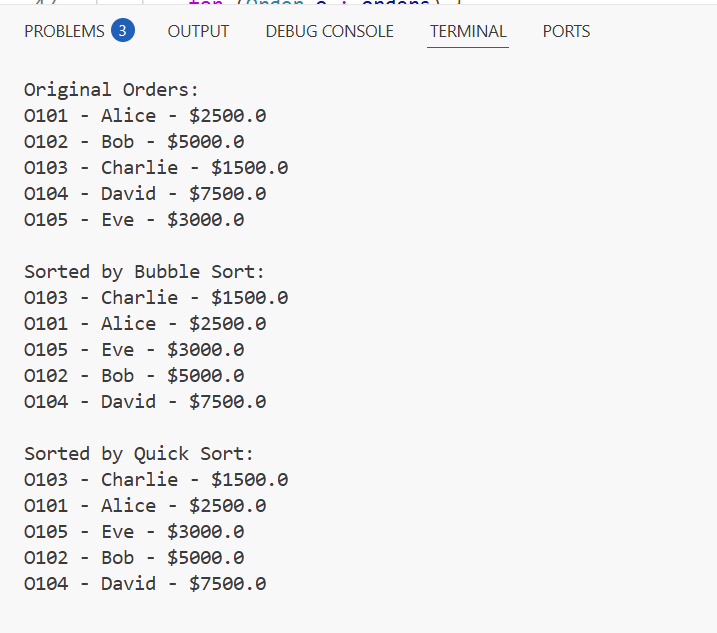
**Order.java:**



**OrderSorter.java:**



**OUTPUT:**



|  |  |  |
| --- | --- | --- |
| Operation | Bubble Sort | Quick Sort |
| Best | O(n) | O(n log n) |
| Average | O(n^2) | O(n log n) |
| Worst | O(n^2) | O(n^2) |

**Why Quick Sort is Preferred**

* Much faster for large datasets.
* Performs well in real-world scenarios.
* Uses **divide and conquer** strategy.
* Lower constant factors than Merge Sort.
* Bubble Sort is **only educational** — it's not practical for real applications.

**Exercise 4: Employee Management System**

**How Arrays Are Represented in Memory:**

* Arrays are stored in **contiguous memory locations**.
* Each element can be accessed directly using an **index**.
* The **base address + (index × size of element)** gives the location of any item.

**Advantages:**

* **Fast access (O(1))** to elements via indexing.
* Minimal overhead compared to dynamic structures.
* Predictable and cache-friendly

**Limitations of Arrays**

* **Fixed size**: Must define max size (e.g., 100) in advance.
* **Slow delete/insert**: Requires shifting elements.
* **Wasted memory**: Unused elements still occupy space.
* **No dynamic resizing**: Unlike ArrayList.

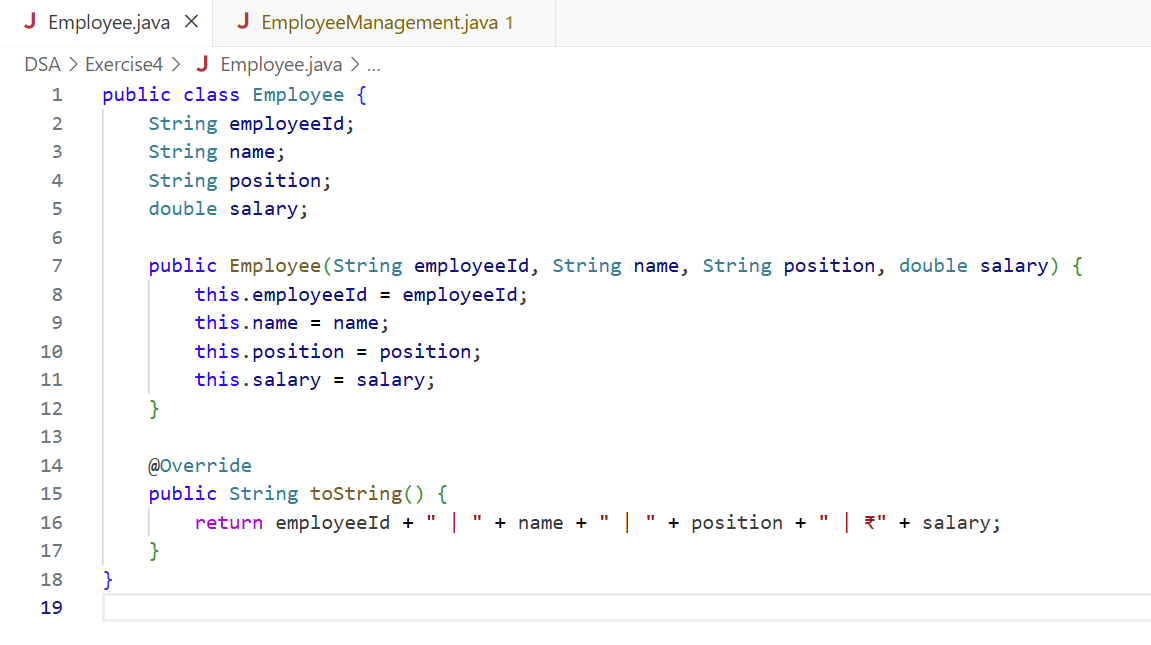
**When to Use Arrays**

* Small, fixed-size datasets.
* You need fast random access (arr[index]).
* Memory is not a constraint.
* Simple programs or teaching tools.

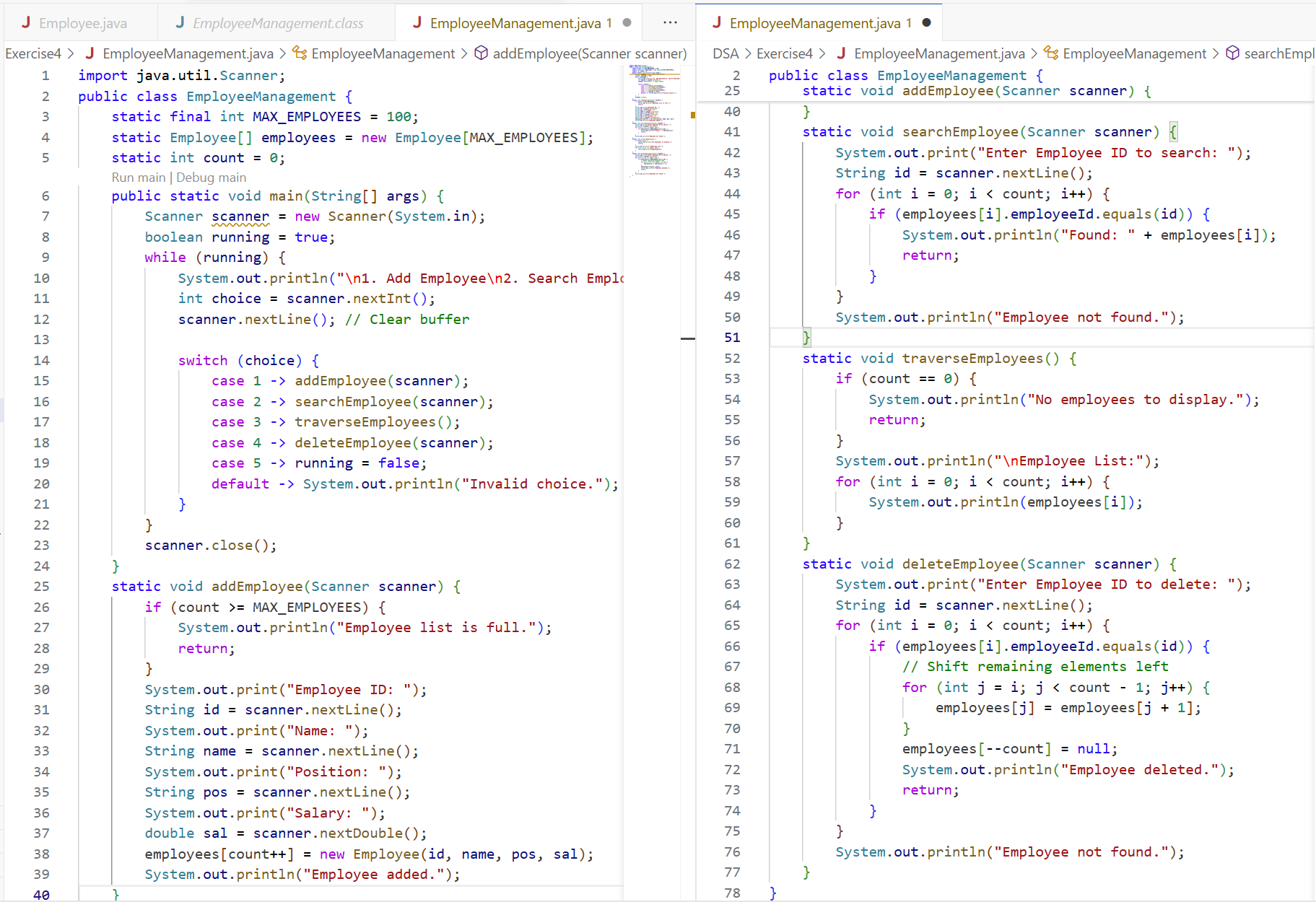
**Time Complexity of Operations (Array-based)**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(1) (at end) | Just place at index count |
| Search | O(n) | Linear scan since unsorted |
| Traverse | O(n) | Iterate from 0 to count |
| Delete | O(n) | Search + shift elements left from deleted idx |

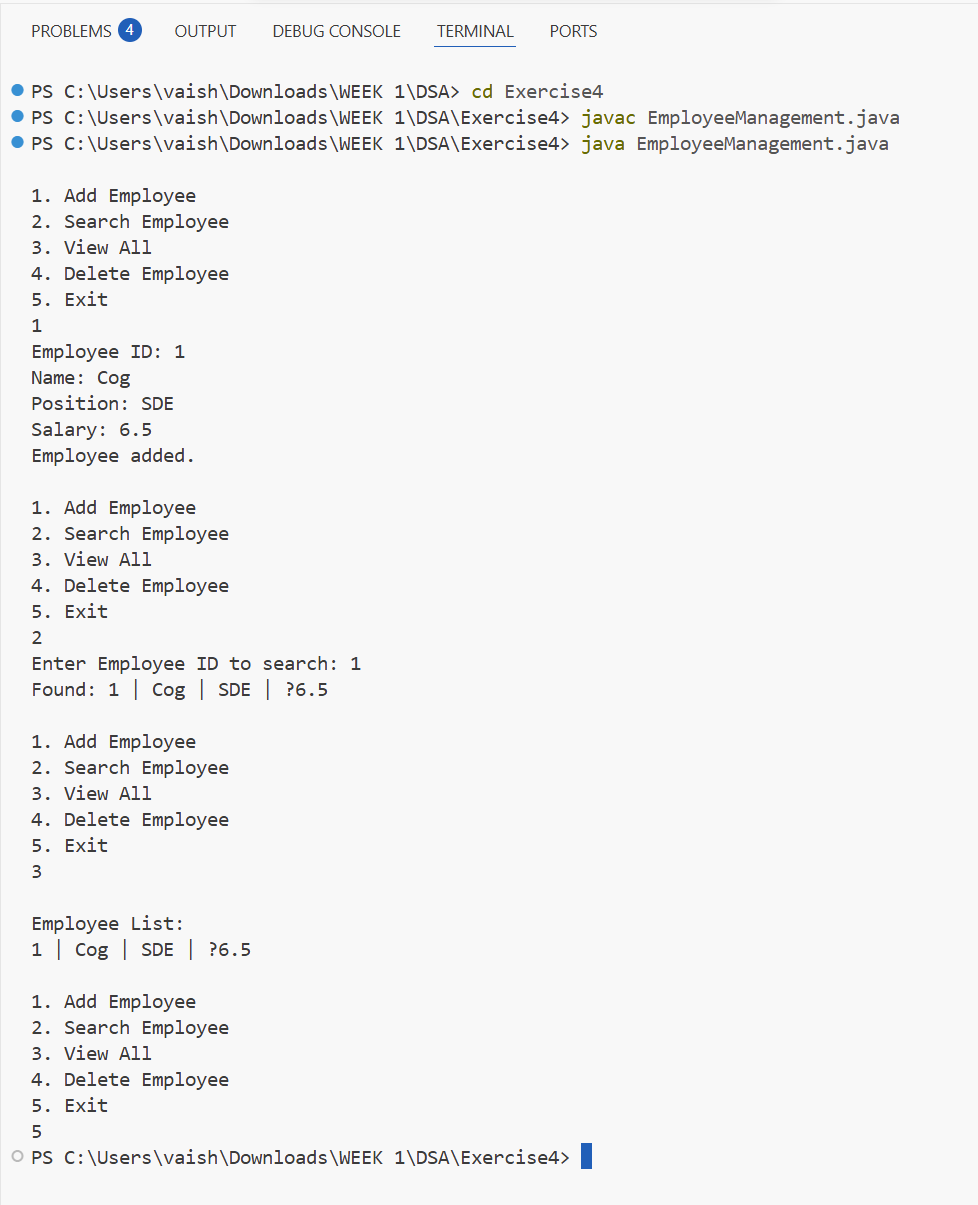
**Employee.java:**



**EmployeeManagement.java:**



**OUTPUT:**



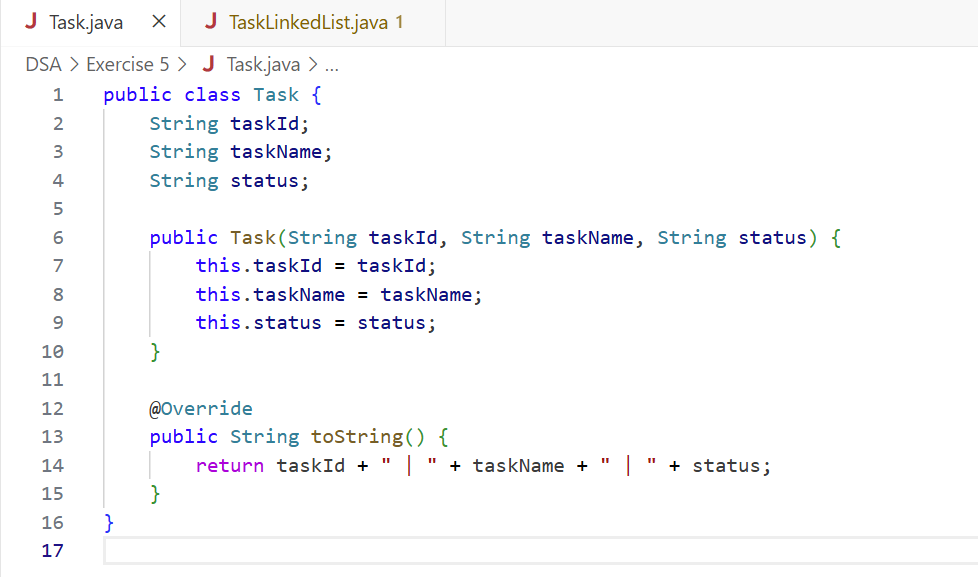
**Exercise 5: Task Management System**

Types of Linked Lists:

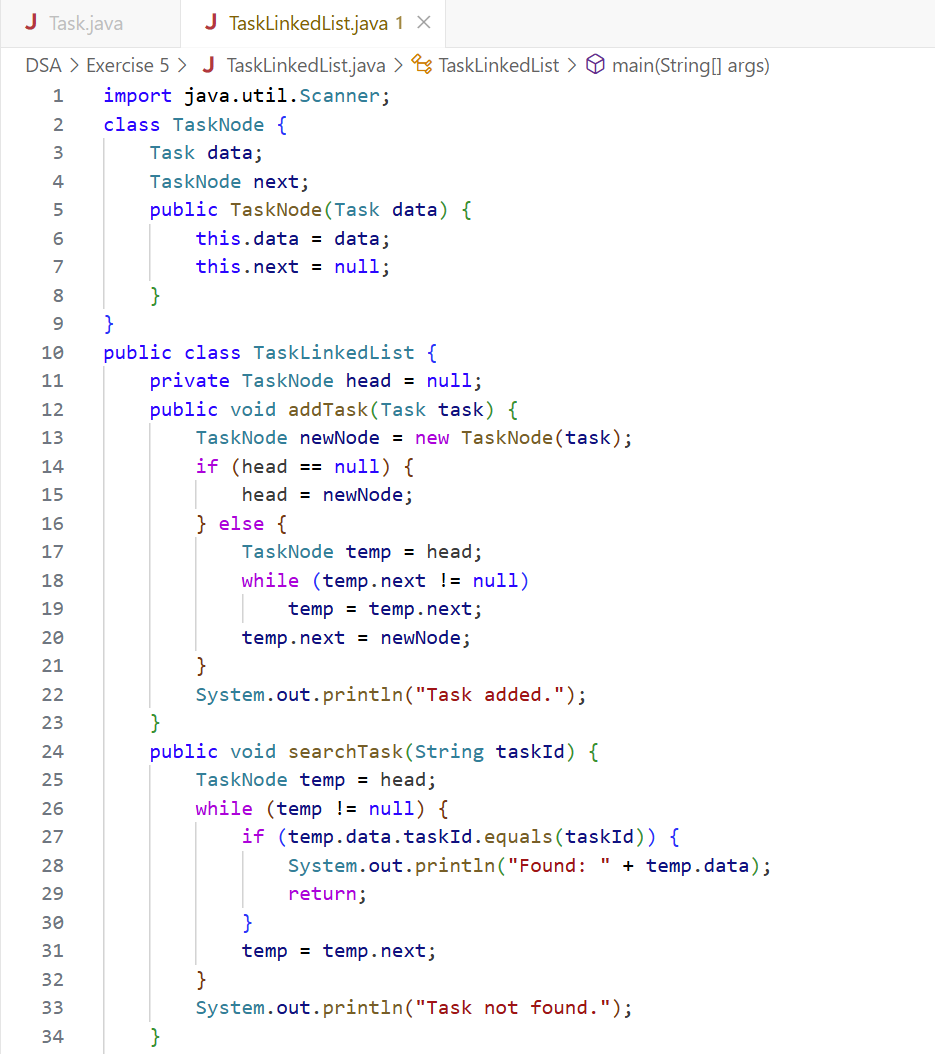
Singly Linked List: Each node has a data field and a pointer to the next node. One-directional.

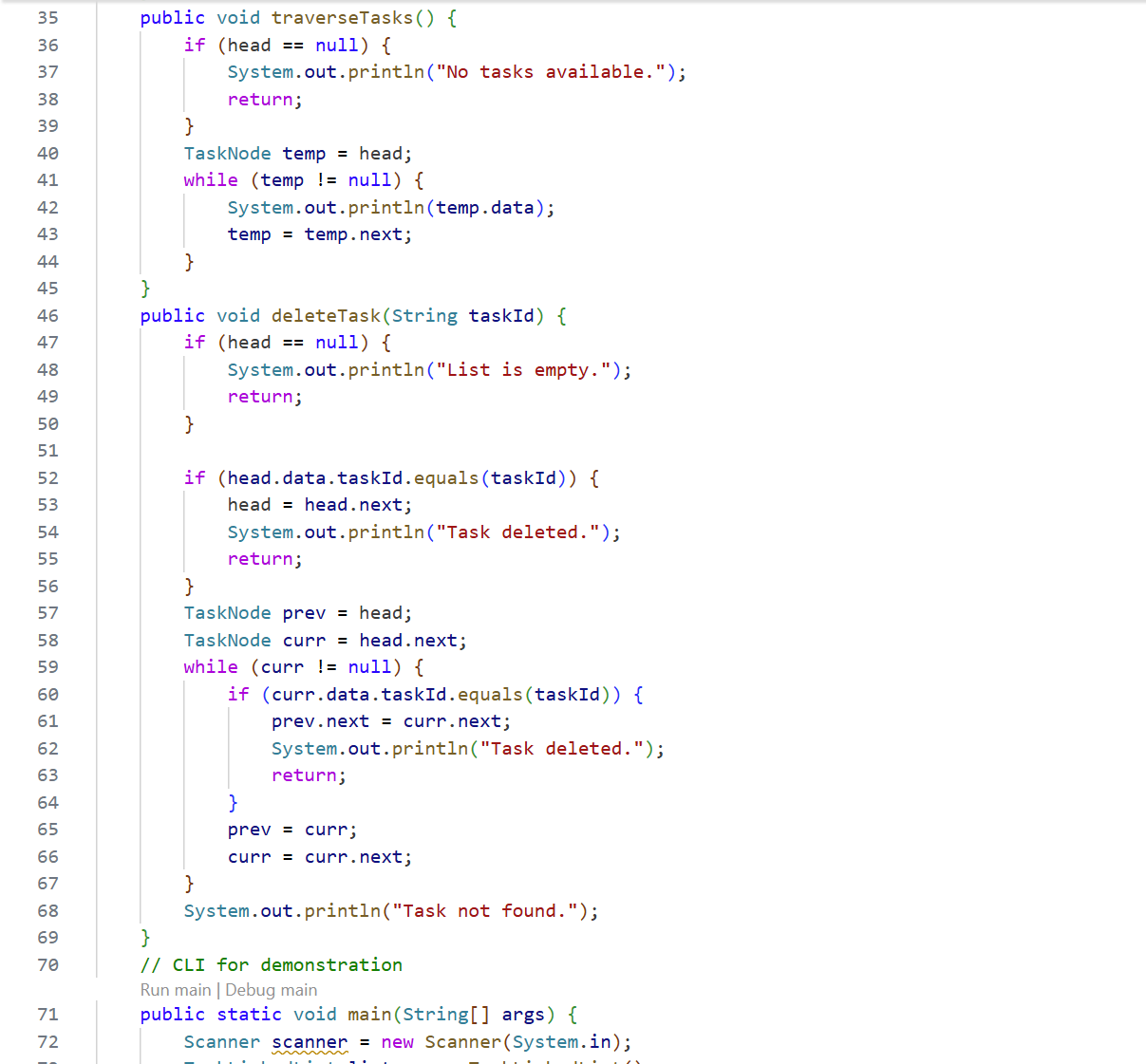
Doubly Linked List: Each node has data, next, and prev. Allows traversal in both directions.

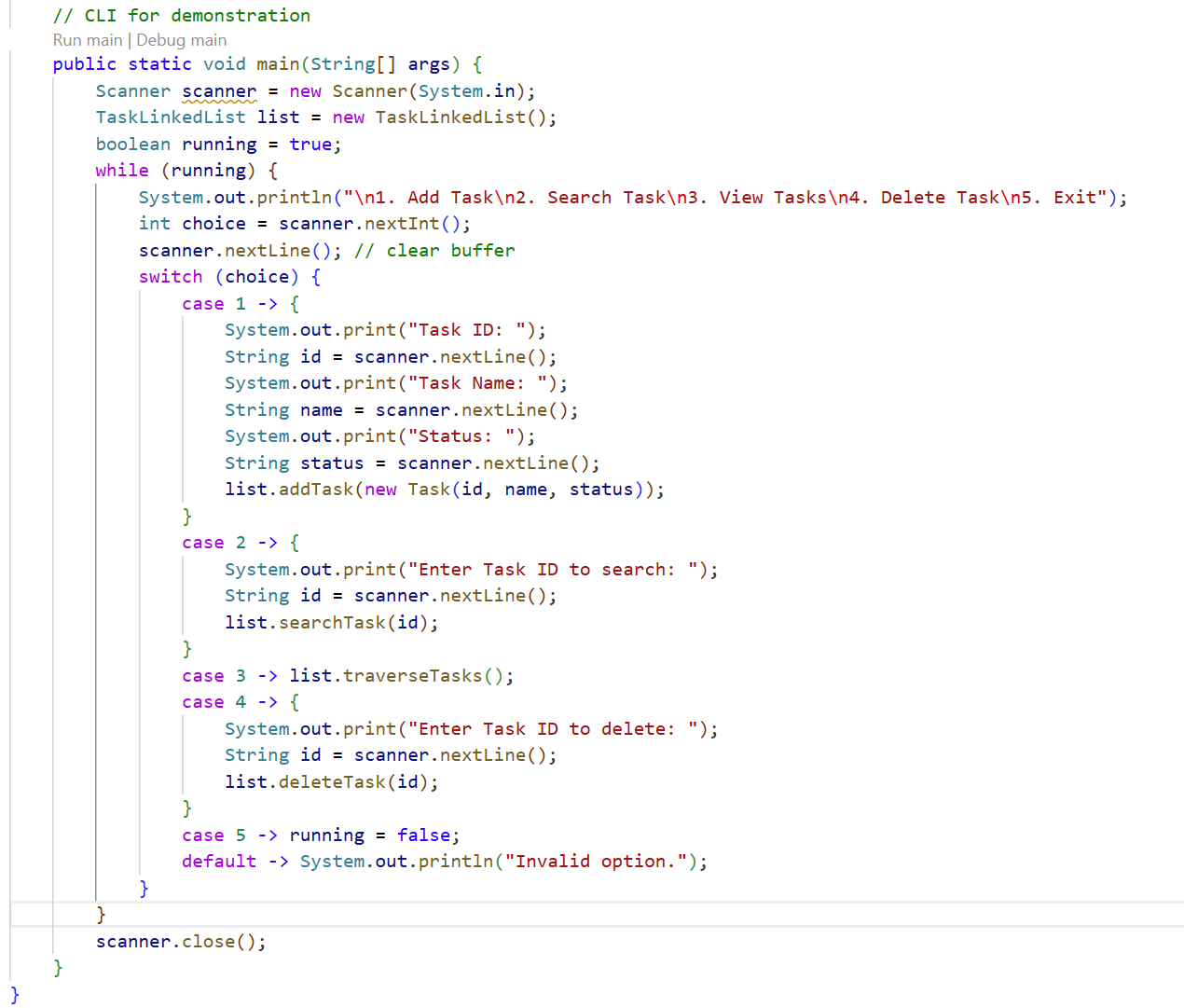
**Task.java:**

****

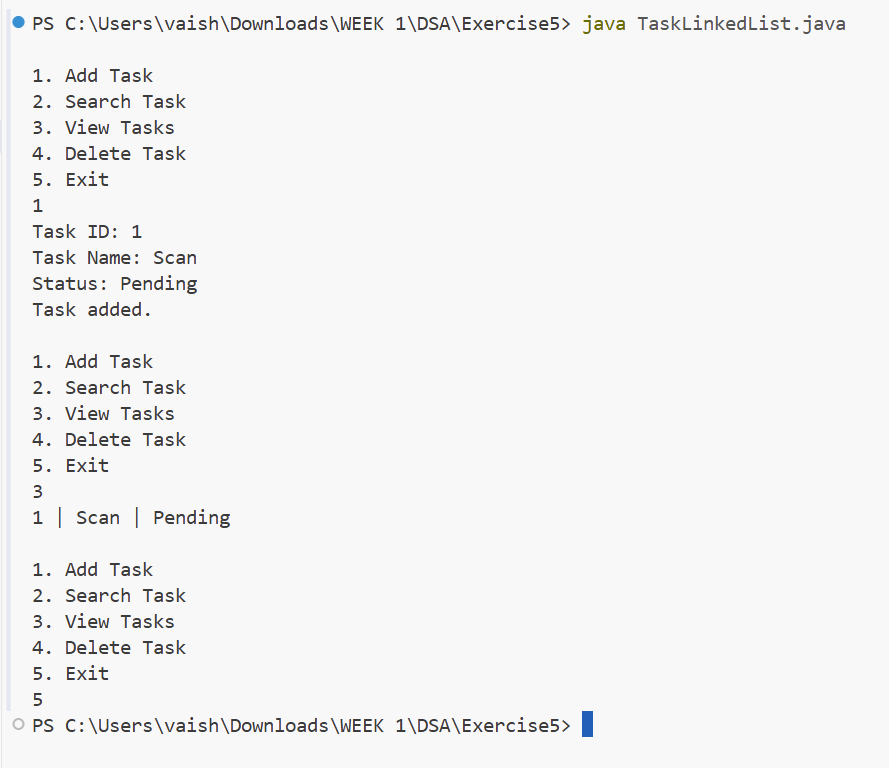
**TaskLinkedList.java:**

****

****

****

**OUTPUT:**

****

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Reason |
| Add | O(n) | Need to traverse to end to insert |
| Search | O(n) | Linear scan through the list |
| Traverse | O(n) | Print each task |
| Delete | O(n) | Need to find node before one to delete |

Advantages of Linked Lists Over Arrays:

* Dynamic Size
* Easier insert/delete (no shifting)
* Efficient memory usage
* No memory reallocation needed

**Exercise 6: Library Management System**

**Linear Search**

* Scans each element one by one until the desired value is found.
* **Works on both sorted and unsorted data.**
* **Time Complexity:**
  + Best: O(1)
  + Worst: O(n)
* **Use Case:** Small or unsorted data.

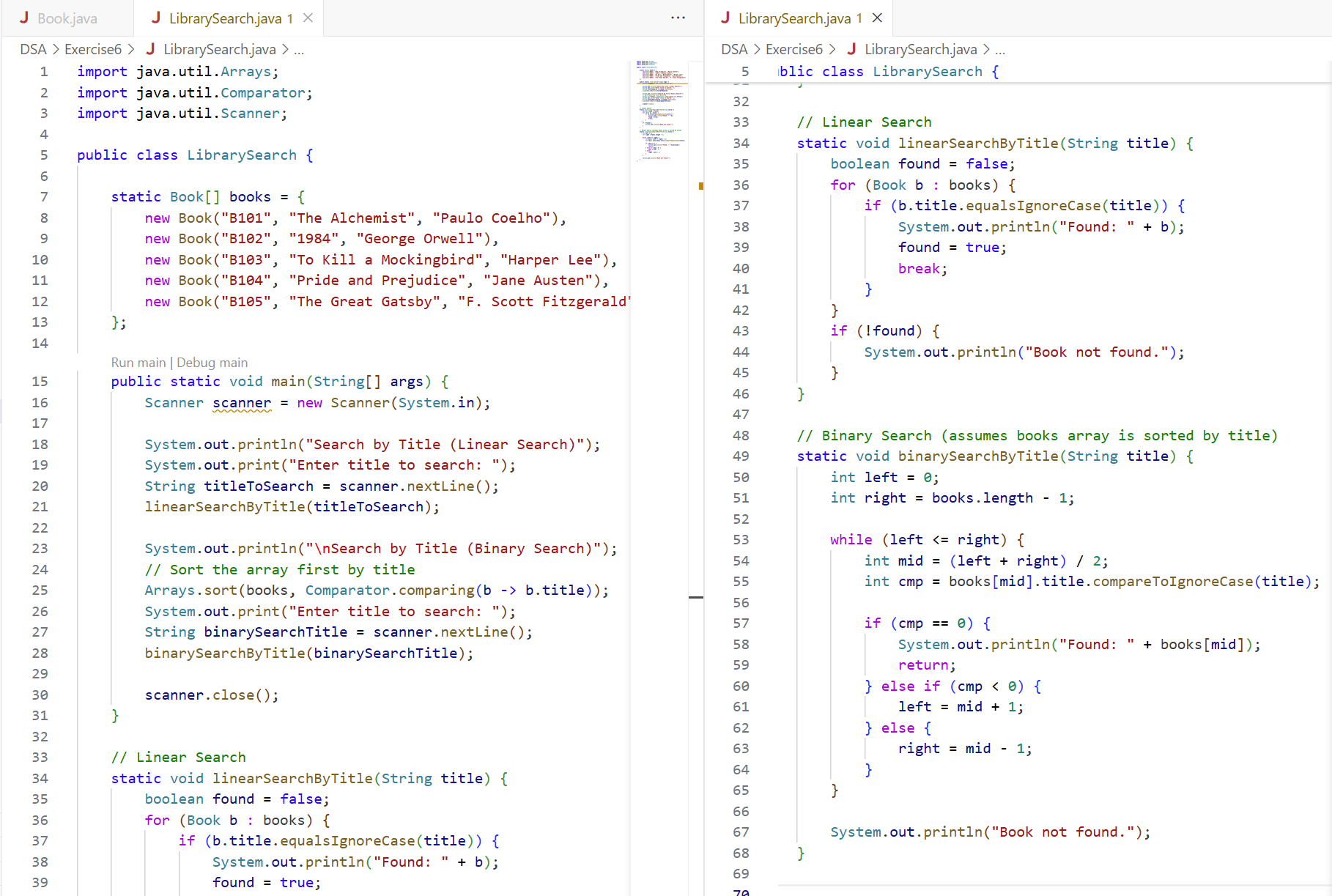
**Binary Search**

* Works on **sorted** data.
* Divides the list in half repeatedly to find the target.
* **Time Complexity:**
  + Best: O(1)
  + Worst: O(log n)
* **Use Case:** Large sorted datasets.

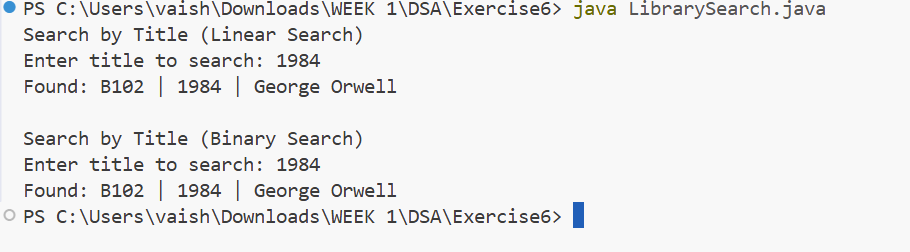
**Book.java:**

****

**LibrarySearch.java:**

****

**OUTPUT:**



**Time Complexity Comparison**

| **Algorithm** | **Best Case** | **Worst Case** |
| --- | --- | --- |
| Linear Search | O(1) | O(n) |
| Binary Search | O(1) | O(log n) |
|  |  |  |