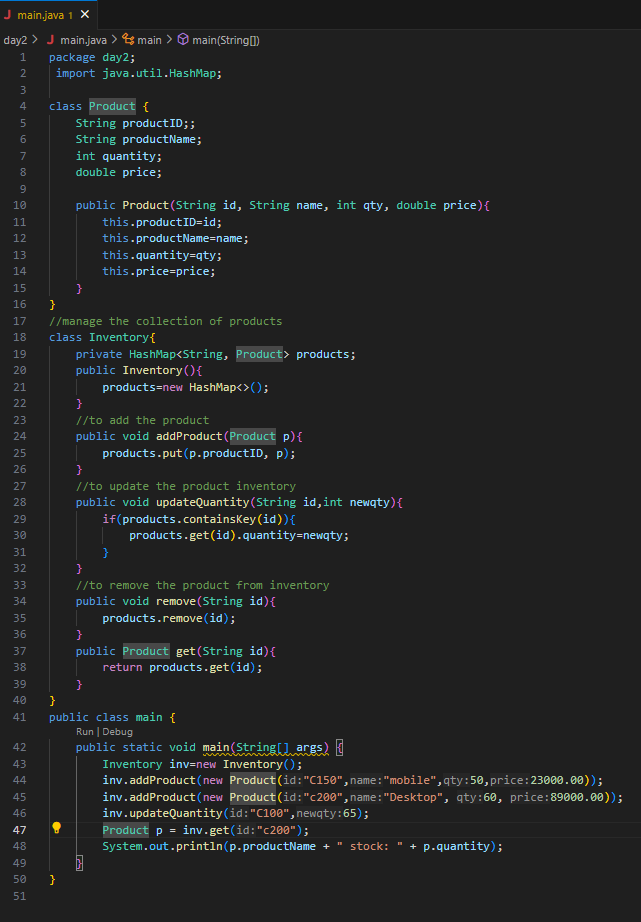
**Exercise 1: Inventory Management System**

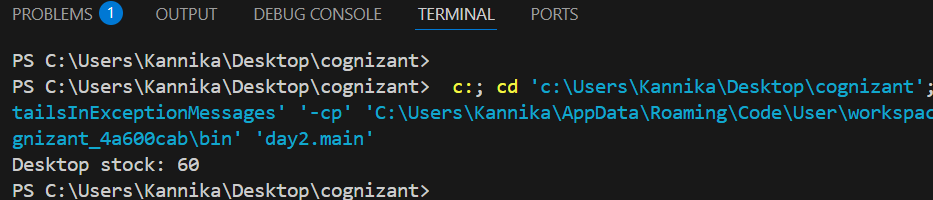
For large inventories, efficient data structures ensure:

* Fast product lookups (O(1) or O(log n) time)
* Quick updates when stock changes
* Memory efficiency when storing thousands of items

Suitable data structures:

* HashMap: Best for quick lookups by product ID (O(1) average case)
* TreeMap: Maintains sorted order (O(log n) operations)
* ArrayList: Simple but slower searches 

OUTPUT:



Time Complexity Analysis

* Add: O(1) - HashMap put operation
* Update: O(1) - HashMap get + field update
* Delete: O(1) - HashMap remove
* Search: O(1) - HashMap get

Optimizations

* Use HashMap for O(1) operations
* For range queries (e.g., "show low stock"), maintain a separate TreeSet sorted by quantity
* Consider concurrent structures if multiple users access inventory simultaneously

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

1. Understanding Asymptotic Notation

Big O Notation

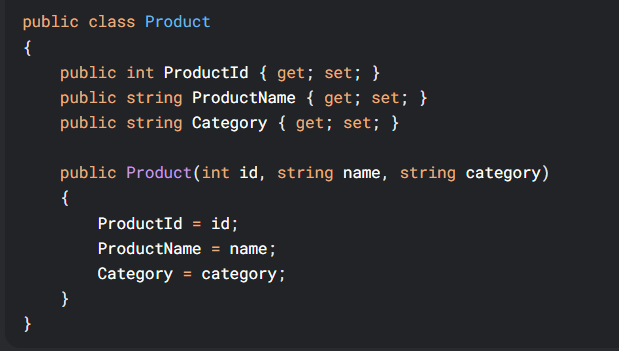
Big O notation describes how an algorithm's runtime or space requirements grow as the input size grows. It provides an upper bound on complexity, helping us compare algorithm efficiency.

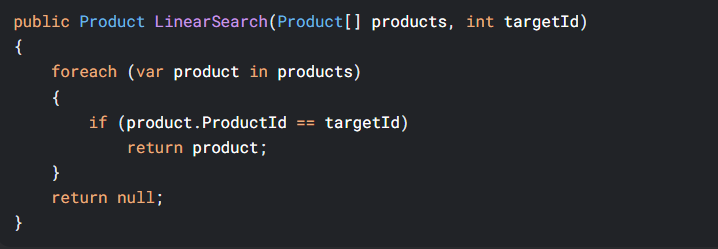
For search operations:

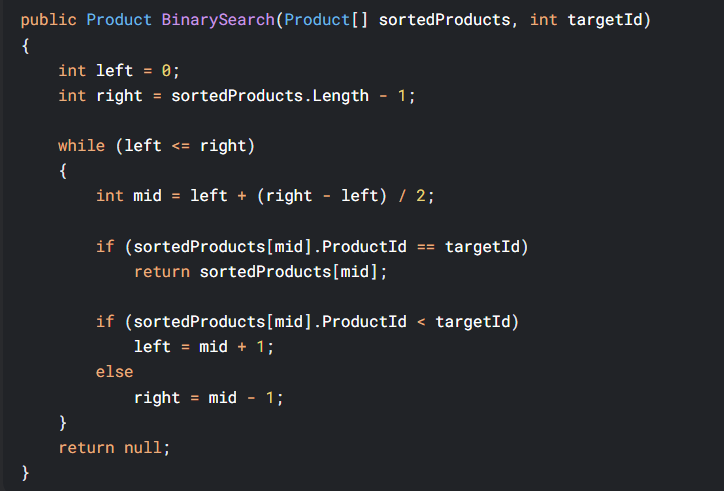
* O(1): Constant time (ideal)
* O(log n): Logarithmic time (excellent)
* O(n): Linear time (fair for small datasets)
* O(n²): Quadratic time (poor)

Search Operation Scenarios

* Best case: Minimum possible time (e.g., finding item on first try)
* Average case: Expected time for random inputs
* Worst case: Maximum possible time (critical for performance guarantees)







Platform Recommendation

Binary search is superior for our e-commerce platform when:

1. Products are sorted by ID (a one-time sorting cost enables faster searches)
2. Large inventory (logarithmic scaling handles growth better)
3. Search-by-ID operations are frequent

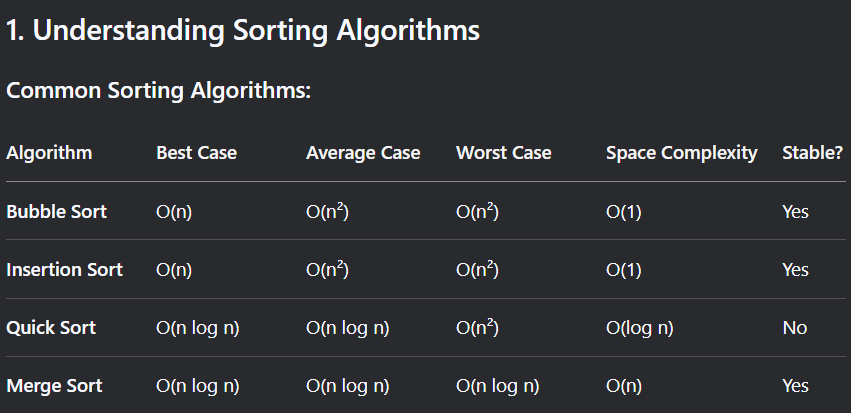
However, consider these real-world factors:

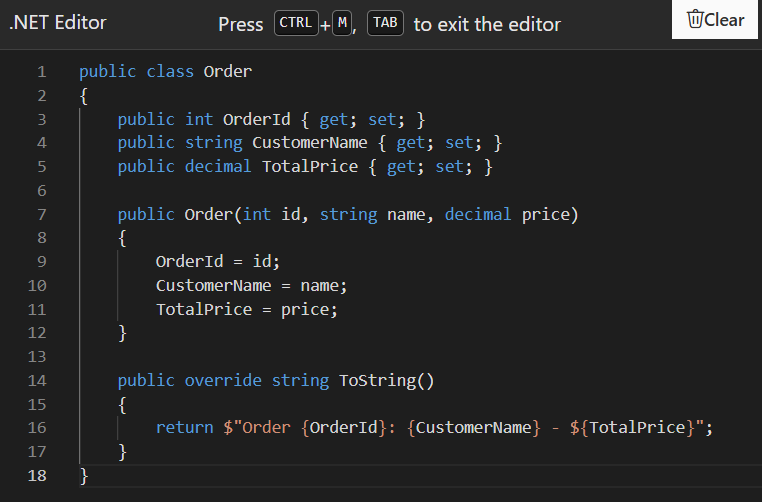
1. Indexing: A database index would be better than in-memory arrays
2. Multiple search criteria: May require more advanced structures (hash tables, search trees)
3. Dynamic inventory: Requires maintaining sorted order (consider a self-balancing BST)

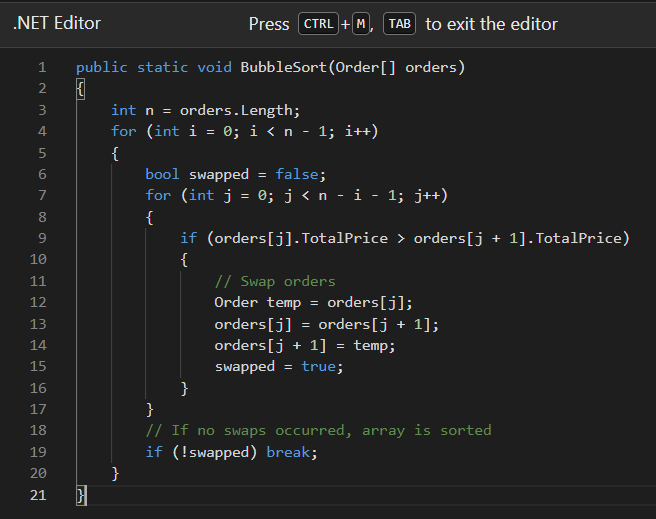
Implementation advice:

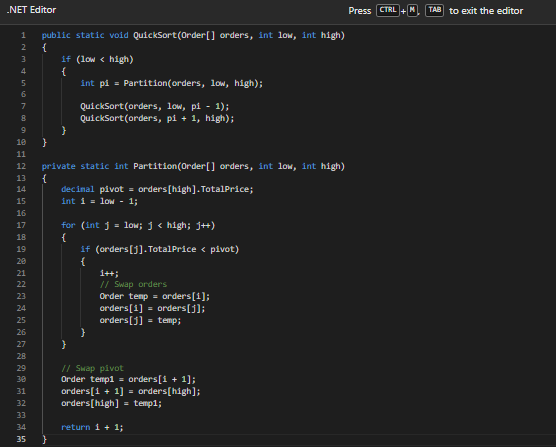
1. Use binary search for primary key (ID) lookups
2. Combine with a hash table for name/category searches
3. Consider database indexing for persistent storage

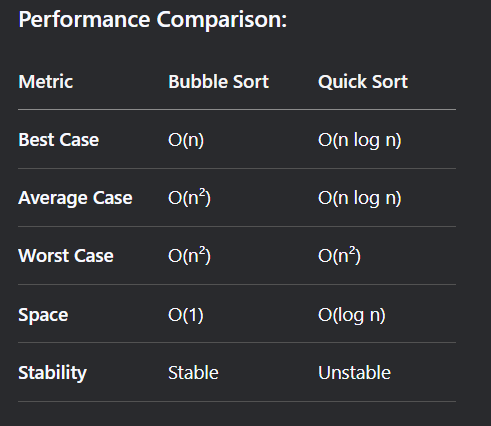
**Exercise 3: Sorting Customer Orders**











**Exercise 4: Employee Management System**

Array Representation in Memory

Memory Representation:

Contiguous block: Array elements are stored in adjacent memory locations

Index-based access: Each element's address = base address + (index × element size)

Fixed capacity: Size must be declared upfront (static memory allocation)

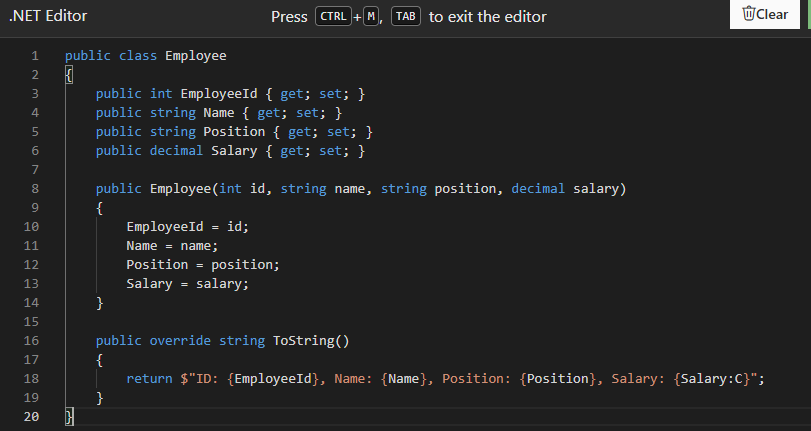
Advantages:

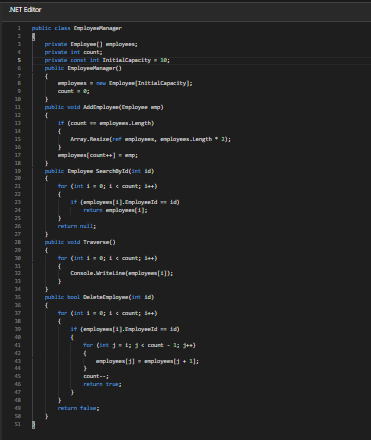
Fast random access - O(1) time for accessing any element by index

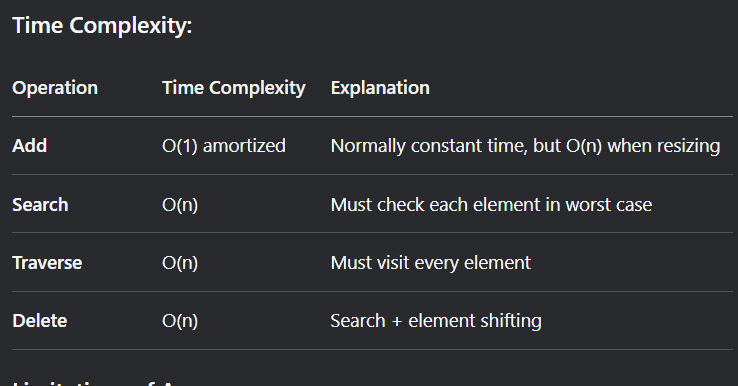
Memory efficiency - No overhead for pointers/links (unlike linked lists)

Cache friendliness - Contiguous memory improves cache performance

Simple implementation - Easy to understand and use







Limitations of Arrays:

Fixed Size:

Must resize (expensive operation) when capacity is exceeded

Solution: Use List<T> in practice which handles resizing automatically

Insertion/Deletion Costs:

Adding/removing in middle requires shifting elements

Solution: Linked lists better for frequent modifications

Memory Waste:

Unused allocated space if capacity > actual count

Solution: Dynamic collections that grow as needed

Single Data Type:

Arrays store only one type

Solution: Use object arrays or generic collections

When to Use Arrays:

Small, fixed-size datasets

When random access is primary operation

Performance-critical sections where cache locality matters

As building blocks for more complex data structures.

**Exercise 5: Task Management System**

## **Understand Linked Lists**

### **Singly Linked List**

A data structure consisting of nodes.

Each node contains **data** and a **reference (pointer) to the next node**.

Only forward traversal is possible.

Example: Node1 -> Node2 -> Node3 -> null

### **Doubly Linked List**

Each node contains **data**, a **pointer to the next node**, and a **pointer to the previous node**.

Allows **forward and backward traversal**.

Requires more memory due to the extra pointer

class Task:

def \_\_init\_\_(self, taskId, taskName, status="Pending"):

self.taskId = taskId

self.taskName = taskName

self.status = status

def \_\_str\_\_(self):

return f"[{self.taskId}] {self.taskName} - {self.status}"

class Node:

def \_\_init\_\_(self, task):

self.task = task

self.next = None

class TaskManager:

def \_\_init\_\_(self):

self.head = None

def add\_task(self, task):

new\_node = Node(task)

if not self.head:

self.head = new\_node

return

current = self.head

while current.next:

current = current.next

current.next = new\_node

def search\_task(self, taskId):

current = self.head

while current:

if current.task.taskId == taskId:

return current.task

current = current.next

return None

def delete\_task(self, taskId):

current = self.head

prev = None

while current:

if current.task.taskId == taskId:

if prev:

prev.next = current.next

else:

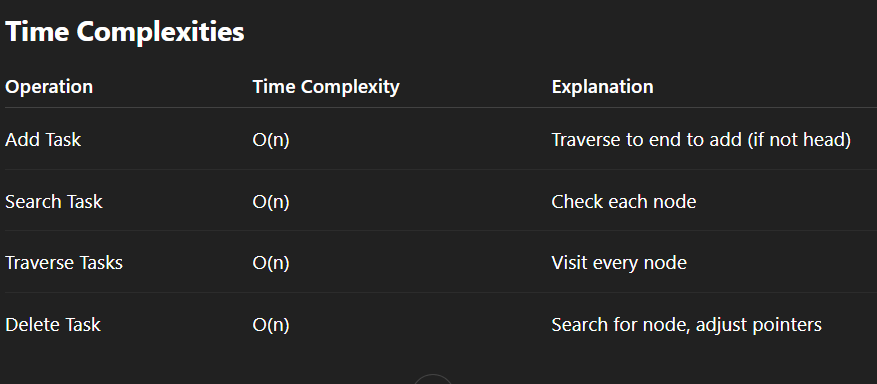
self.head = current.next

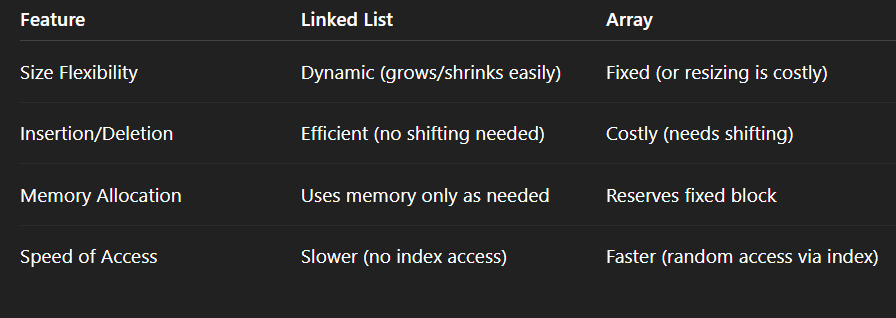
return True

prev = current

current = current.next

return False





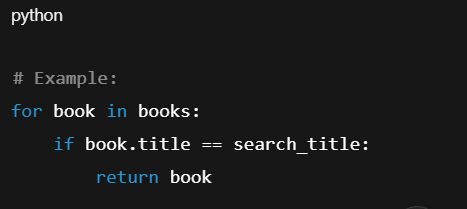
**Exercise 6: Library Management System**

Linear Search

How it works: Goes through each element one by one until it finds a match or reaches the end.

Best case: First item matches.

Worst case: Item is not found or at the end of the list.



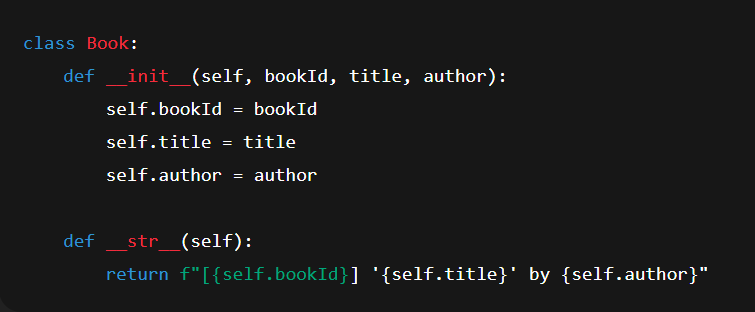
Binary Search

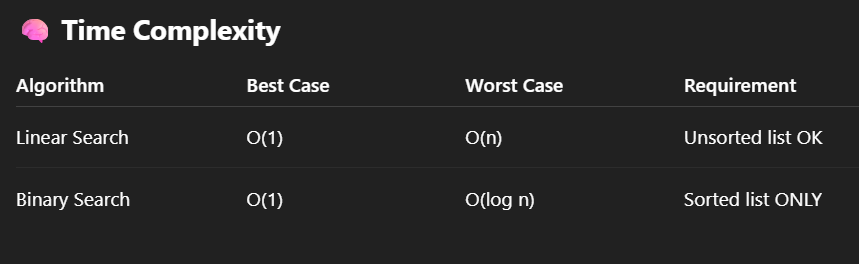
How it works: Repeatedly divides a sorted list in half to locate the item.

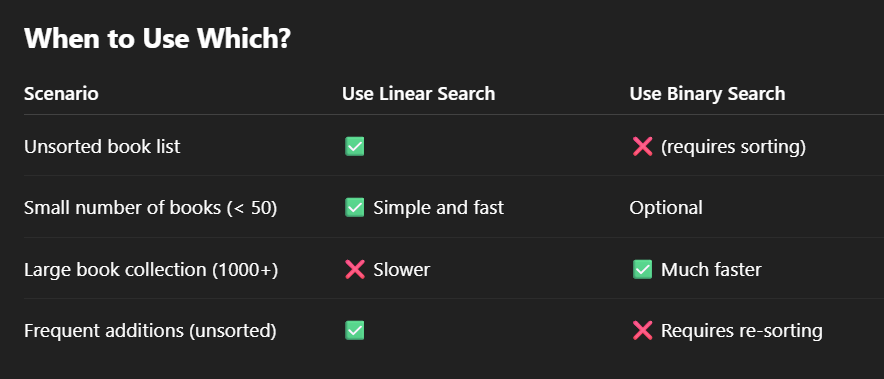
Requirement: Data must be sorted (e.g., alphabetically by title).

Best case: Item found at the middle.

Worst case: Logarithmic divisions until found or not.



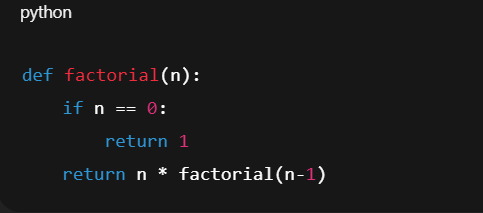


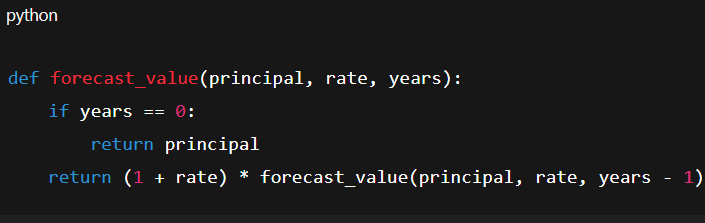


**Exercise 7: Financial Forecasting**

Recursion is a programming technique where a function calls itself to solve a smaller subproblem of the original problem.

Each call reduces the problem, and recursion ends when a base case is reached.





Time Complexity

Each recursive call reduces n by 1 → Total of n recursive calls.

Time Complexity: O(n)

Space Complexity

Due to call stack, recursion takes O(n) space.