Sure, here's an expanded version with more details on how each system works and the associated Java code for each exercise.

---

# Inventory Management System

## Scope

This system allows managing an inventory of products, including adding, updating, deleting, and retrieving products. Each product has a unique ID, name, quantity, and price.

## Big O Notation

- \*\*Add Product:\*\* O(1)

- \*\*Update Product:\*\* O(1)

- \*\*Delete Product:\*\* O(1)

- \*\*Get Product:\*\* O(1)

## How It Works

The inventory is managed using a `HashMap` where the key is the product ID, and the value is the product object. This allows for efficient retrieval, addition, updating, and deletion operations.

## Implementation

### Product Class

```java

class Product {

int productId;

String productName;

int quantity;

double price;

Product(int productId, String productName, int quantity, double price) {

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

@Override

public String toString() {

return "Product ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: " + price;

}

}

```

### Inventory Management Class

```java

import java.util.HashMap;

public class InventoryManagement {

private HashMap<Integer, Product> inventory;

public InventoryManagement() {

inventory = new HashMap<>();

}

public void addProduct(Product product) {

inventory.put(product.productId, product);

System.out.println("Product added: " + product);

}

public void updateProduct(Product product) {

if (inventory.containsKey(product.productId)) {

inventory.put(product.productId, product);

System.out.println("Product updated: " + product);

} else {

System.out.println("Product not found: " + product.productId);

}

}

public void deleteProduct(int productId) {

Product removedProduct = inventory.remove(productId);

if (removedProduct != null) {

System.out.println("Product removed: " + removedProduct);

} else {

System.out.println("Product not found: " + productId);

}

}

public Product getProduct(int productId) {

return inventory.get(productId);

}

public static void main(String[] args) {

InventoryManagement inventory = new InventoryManagement();

Product product1 = new Product(1, "Laptop", 10, 999.99);

Product product2 = new Product(2, "Smartphone", 20, 499.99);

inventory.addProduct(product1);

inventory.addProduct(product2);

System.out.println("Get Product ID 1: " + inventory.getProduct(1));

product1.quantity = 15;

inventory.updateProduct(product1);

System.out.println("Get Updated Product ID 1: " + inventory.getProduct(1));

inventory.deleteProduct(2);

System.out.println("Get Product ID 2: " + inventory.getProduct(2));

}

}

```

---

# E-commerce Platform Search Function

## Scope

This function enables searching for products on an e-commerce platform by name using linear and binary search. Linear search is used for unsorted arrays, while binary search is used for sorted arrays.

## Big O Notation

- \*\*Linear Search:\*\* O(n)

- \*\*Binary Search:\*\* O(log n)

## How It Works

- \*\*Linear Search:\*\* Iterates through the list of products and compares each product's name with the search query.

- \*\*Binary Search:\*\* Requires the list to be sorted. It repeatedly divides the search interval in half until the product is found or the interval is empty.

## Implementation

### Product Class

```java

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

```

### Search Functions

```java

public class ECommerceSearch {

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

for (int i = 0; i < products.length; i++) {

if (products[i].productName.equals(productName)) {

return i;

}

}

return -1;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (comparison == 0) {

return mid;

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Smartphone", "Electronics"),

new Product(3, "Book", "Books")

};

System.out.println("Linear Search for 'Laptop': " + linearSearch(products, "Laptop"));

System.out.println("Binary Search for 'Book': " + binarySearch(products, "Book"));

}

}

```

---

# Sorting Customer Orders

## Scope

This exercise involves sorting customer orders based on their total prices using Bubble Sort and Quick Sort algorithms.

## Big O Notation

- \*\*Bubble Sort:\*\* O(n^2)

- \*\*Quick Sort:\*\* O(n log n) on average, O(n^2) in the worst case

## How It Works

- \*\*Bubble Sort:\*\* Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process repeats until the list is sorted.

- \*\*Quick Sort:\*\* Divides the list into smaller sub-lists based on a pivot element. Elements less than the pivot go to the left, and elements greater go to the right. This process recursively sorts the sub-lists.

## Implementation

### Order Class

```java

class Order {

int orderId;

String customerName;

double totalPrice;

Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

```

### Sorting Functions

```java

public class OrderSorting {

public static void bubbleSort(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = (low - 1);

for (int j = low; j < high; j++) {

if (orders[j].totalPrice <= pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 300.50),

new Order(2, "Bob", 150.75),

new Order(3, "Charlie", 200.20)

};

bubbleSort(orders);

System.out.println("Orders sorted by Bubble Sort:");

for (Order order : orders) {

System.out.println(order.customerName + ": " + order.totalPrice);

}

quickSort(orders, 0, orders.length - 1);

System.out.println("Orders sorted by Quick Sort:");

for (Order order : orders) {

System.out.println(order.customerName + ": " + order.totalPrice);

}

}

}

```

---

**# Employee Management System**

## Scope

This system involves managing employee records, allowing the addition, search, traversal, and deletion of employees. Each employee has a unique ID, name, position, and salary.

## Big O Notation

- \*\*Add Employee:\*\* O(1)

- \*\*Search Employee:\*\* O(n)

- \*\*Traverse Employees:\*\* O(n)

- \*\*Delete Employee:\*\* O(n)

## How It Works

Employees are managed using an array. The system provides methods for adding new employees, searching for an employee by ID, traversing the list of employees, and deleting an employee by ID.

## Implementation

### Employee Class

```java

class Employee {

int employeeId;

String name;

String position;

double salary;

Employee(int employeeId, String name, String position, double salary) {

this.employeeId

= employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

}

### Employee Management Class

import java.util.ArrayList;

public class EmployeeManagement {

private ArrayList<Employee> employees;

public EmployeeManagement() {

employees = new ArrayList<>();

}

public void addEmployee(Employee employee) {

employees.add(employee);

System.out.println("Employee added: " + employee.name);

}

public Employee searchEmployee(int employeeId) {

for (Employee employee : employees) {

if (employee.employeeId == employeeId) {

return employee;

}

}

return null;

}

public void traverseEmployees() {

for (Employee employee : employees) {

System.out.println("Employee ID: " + employee.employeeId +

", Name: " + employee.name +

", Position: " + employee.position +

", Salary: " + employee.salary);

}

}

public void deleteEmployee(int employeeId) {

Employee toRemove = null;

for (Employee employee : employees) {

if (employee.employeeId == employeeId) {

toRemove = employee;

break;

}

}

if (toRemove != null) {

employees.remove(toRemove);

System.out.println("Employee removed: " + toRemove.name);

} else {

System.out.println("Employee not found: " + employeeId);

}

}

public static void main(String[] args) {

EmployeeManagement employeeManagement = new EmployeeManagement();

Employee emp1 = new Employee(1, "John Doe", "Manager", 80000);

Employee emp2 = new Employee(2, "Jane Smith", "Developer", 60000);

employeeManagement.addEmployee(emp1);

employeeManagement.addEmployee(emp2);

System.out.println("Searching for Employee ID 1: " + employeeManagement.searchEmployee(1).name);

System.out.println("Traversing Employees:");

employeeManagement.traverseEmployees();

employeeManagement.deleteEmployee(2);

System.out.println("Traversing Employees after deletion:");

employeeManagement.traverseEmployees();

}

}