**Problem Understanding:**

In an e-commerce platform where customer orders are being received continuously. Implementation has two sorting algorithms: Bubble Sort and Quick Sort.

**1. Bubble Sort**

* Repeatedly compares adjacent elements and swaps them if they are in the wrong order.
* Time Complexity:
  + Best: O(n)
  + Average/Worst: O(n²)
* Easy to implement but inefficient for large datasets.

**2. Insertion Sort**

* Builds the sorted list one item at a time by inserting each element into its correct position.
* Time Complexity:
  + Best: O(n)
  + Average/Worst: O(n²)
* Efficient for small or nearly sorted datasets.

**3. Quick Sort**

* Uses a pivot to divide the array into smaller parts and recursively sorts them.
* Time Complexity:
  + Best/Average: O(n log n)
  + Worst: O(n²)
* Very fast for large datasets, widely used.

**4. Merge Sort**

* Divides the array into halves, sorts them, and merges back.
* Time Complexity:
  + Best/Average/Worst: O(n log n)
* Stable and predictable, great for linked lists or external sorting.

**Implementation:**

class Order {

String orderId, customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String toString() {

return "Order ID: " + orderId + ", Customer: " + customerName + ", Total Price: " + totalPrice;

}

}  
  
public class OrderSearch {

public static void main(String[] args) {

Order[] orders = {new Order("O001", "Alice", 450.00),

new Order("O002", "Bob", 120.00),

new Order("O003", "Charlie", 300.50),

new Order("O004", "David", 220.75),

new Order("O005", "Eve", 510.00)};

System.***out***.println("Before Bubble Sort:");

*printOrders*(orders);

*bubbleSort*(orders);

System.***out***.println("\nAfter Bubble Sort (by total price):");

*printOrders*(orders);

Order[] orders2 = {new Order("O001", "Alice", 450.00),

new Order("O002", "Bob", 120.00),

new Order("O003", "Charlie", 300.50),

new Order("O004", "David", 220.75),

new Order("O005", "Eve", 510.00)};

System.***out***.println("\nBefore Quick Sort:");

*printOrders*(orders2);

*quickSort*(orders2, 0, orders2.length - 1);

System.***out***.println("\nAfter Quick Sort (by total price):");

*printOrders*(orders2);

}

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 printOrders(Order[] orders) {

for (Order o : orders) {

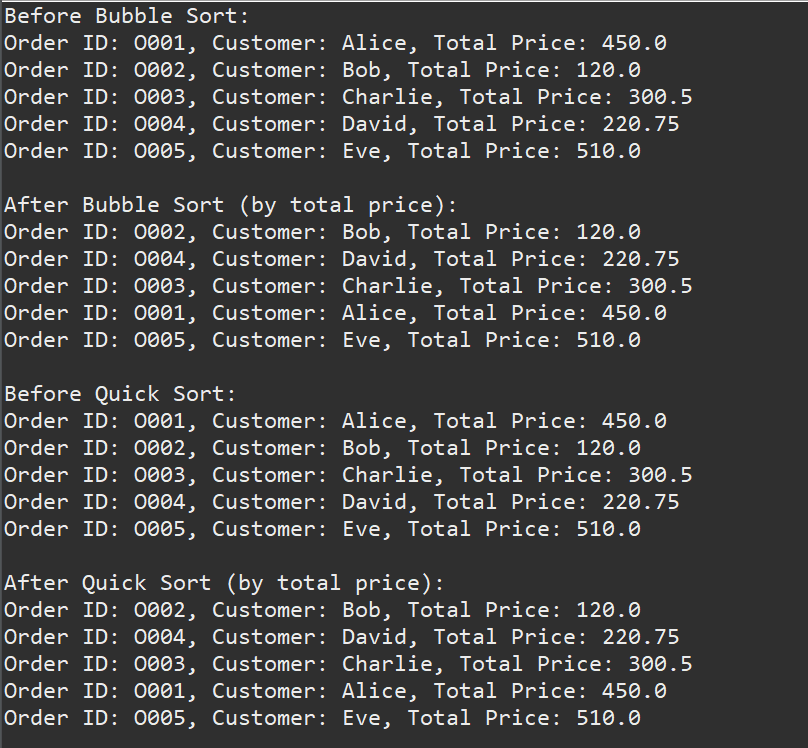
System.***out***.println(o);

}

}

}

**Output:**

****

**Analysis:**

|  |  |  |
| --- | --- | --- |
| Feature / Case | Bubble Sort | Quick Sort |
| Best Case | O(n) – already sorted | O(n log n) – pivot divides evenly |
| Average Case | O(n²) – many swaps | O(n log n) – divide and conquer |
| Worst Case | O(n²) – reverse order | O(n²) – unbalanced pivot (rare) |

**Why Quick Sort is generally preferred over Bubble Sort?**

* Quick Sort is preferred over Bubble Sort because it is faster and more efficient, especially for large datasets.
* It uses a divide-and-conquer approach, making it highly scalable.
* Bubble Sort is simple but slow due to its O(n²) time complexity.
* Quick Sort is commonly used in real-world libraries and applications.
* For sorting customer orders in e-commerce, Quick Sort is the better choice.