**Exercise 3: Sorting Customer Orders**

**Understanding Sorting Algorithms**

Efficient sorting is crucial in many real-world applications such as prioritizing high-value customer orders, especially in e-commerce platforms. There are various sorting algorithms, each with its own strengths and weaknesses. Let’s briefly explore four key ones.

**🔹 Bubble Sort**

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process continues until no more swaps are needed, indicating the list is sorted.

* **Best use case**: Educational purposes or very small datasets.
* **Time complexity**: O(n²) in all cases (best, average, worst).
* **Space complexity**: O(1) – In-place sorting.
* **Performance**: Very slow for large datasets due to repeated passes.

**🔹 Insertion Sort**

Insertion Sort builds the final sorted array one element at a time. It removes one element from the input, finds the location it belongs to in the sorted list, and inserts it there. It is more efficient than bubble sort on small or nearly sorted arrays.

* **Best use case**: Small datasets or partially sorted data.
* **Time complexity**: O(n²) average and worst, O(n) best case (when already sorted).
* **Space complexity**: O(1).

**🔹 Merge Sort**

Merge Sort is a divide-and-conquer algorithm. It divides the array into halves, sorts them recursively, and then merges the sorted halves.

* **Best use case**: Very large datasets or linked lists.
* **Time complexity**: O(n log n) in all cases.
* **Space complexity**: O(n) – uses auxiliary space for merging.
* **Performance**: Stable and predictable, good for external sorting.

**🔹 Quick Sort**

Quick Sort is another divide-and-conquer algorithm. It selects a pivot element, partitions the array so elements less than the pivot are on one side and greater ones on the other, and recursively sorts the partitions.

* **Best use case**: General-purpose in-memory sorting.
* **Time complexity**: O(n log n) on average, O(n²) in worst case.
* **Space complexity**: O(log n) for recursion.
* **Performance**: Faster in practice due to in-place sorting and good cache performance.

**Source Code**

**Order.java**

public class Order {

private final String orderId;

private final String customerName;

private final double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String getOrderId() { return orderId; }

public String getCustomerName() { return customerName; }

public double getTotalPrice() { return totalPrice; }

@Override

public String toString() {

return orderId + " | " + customerName + " | " + totalPrice;

}

}

**OrderSorter.java**

public class OrderSorter {

public static void bubbleSort(Order[] arr) {

int n = arr.length;

boolean swapped;

do {

swapped = false;

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

if (arr[i - 1].getTotalPrice() < arr[i].getTotalPrice()) {

Order temp = arr[i - 1];

arr[i - 1] = arr[i];

arr[i] = temp;

swapped = true;

}

}

n--;

} while (swapped);

}

public static void quickSort(Order[] arr) {

quick(arr, 0, arr.length - 1);

}

private static void quick(Order[] a, int low, int high) {

if (low < high) {

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

quick(a, low, pi - 1);

quick(a, pi + 1, high);

}

}

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

double pivot = a[high].getTotalPrice();

int i = low;

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

if (a[j].getTotalPrice() > pivot) {

Order temp = a[i];

a[i] = a[j];

a[j] = temp;

i++;

}

}

Order temp = a[i];

a[i] = a[high];

a[high] = temp;

return i;

}

}

**Demo.java**

public class Demo {

public static void main(String[] args) {

Order[] orders = {

new Order("O101", "Ankit", 2499.99),

new Order("O102", "Meera", 899.50),

new Order("O103", "Ravi", 5400.00),

new Order("O104", "Priya", 1200.00),

new Order("O105", "Sahil", 3100.75)

};

// Bubble Sort

Order[] bubbleSorted = orders.clone();

OrderSorter.bubbleSort(bubbleSorted);

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

for (Order o : bubbleSorted) System.out.println(o);

// Quick Sort

Order[] quickSorted = orders.clone();

OrderSorter.quickSort(quickSorted);

System.out.println("\nOrders sorted using Quick Sort:");

for (Order o : quickSorted) System.out.println(o);

}

}

**Output:**

**A computer screen shot of a black screen

AI-generated content may be incorrect.**

**Analysis**

**🔸 Performance Comparison**

* **Bubble Sort** has a time complexity of O(n²) in all cases. It performs many redundant comparisons and swaps, making it inefficient for large datasets.
* **Quick Sort**, on the other hand, performs much better with average and best-case time complexity of O(n log n). Though its worst case is O(n²), this can be avoided with good pivot selection (e.g., choosing a random or median pivot).

**🔸 Memory Usage**

* Bubble Sort is in-place and uses O(1) extra space.
* Quick Sort is also in-place but uses O(log n) space due to recursive calls.

**🔸 Real-World Preference**

Quick Sort is generally preferred in real-world applications because of its efficiency and practical speed, especially for large datasets. Modern programming languages and libraries often implement Quick Sort or hybrid variants because:

* It minimizes comparisons and swaps.
* It utilizes memory cache effectively.
* It handles average data distributions very well.

Bubble Sort, while conceptually simple, is not used in production systems due to its poor scalability.

**Conclusion**

Sorting customer orders by their total price is crucial for prioritizing high-value transactions. Although Bubble Sort can demonstrate the basic idea of sorting, Quick Sort is a significantly better choice in terms of speed and efficiency. For professional applications, always prefer Quick Sort or other advanced algorithms like Merge Sort or TimSort.