Sorting Customer Orders by Total Price

1. Understanding Sorting Algorithms

Common Sorting Algorithms:

1. **Bubble Sort**:
   * Repeatedly steps through the list, compares adjacent elements and swaps them if in wrong order
   * Time Complexity: O(n²) in worst and average cases
   * Space Complexity: O(1) (in-place sorting)
   * Stable: Yes (maintains relative order of equal elements)
2. **Insertion Sort**:
   * Builds the final sorted array one item at a time
   * Time Complexity: O(n²) in worst case, O(n) for nearly sorted data
   * Space Complexity: O(1)
   * Stable: Yes
3. **Quick Sort**:
   * Divide-and-conquer algorithm that partitions the array and recursively sorts the partitions
   * Time Complexity: O(n log n) average case, O(n²) worst case
   * Space Complexity: O(log n) due to recursion stack
   * Stable: No (typical implementations)
4. **Merge Sort**:
   * Divide-and-conquer algorithm that divides input into two halves, sorts them, and merges
   * Time Complexity: O(n log n) in all cases
   * Space Complexity: O(n) (not in-place)
   * Stable: Yes

2. Implementation

Order Class:

class Order {

private String orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

// Getters

public String getOrderId() { return orderId; }

public String getCustomerName() { return customerName; }

public double getTotalPrice() { return totalPrice; }

@Override

public String toString() {

return String.format("Order[%s: %s - $%.2f]", orderId, customerName, totalPrice);

}

}

Bubble Sort Implementation:

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].getTotalPrice() < orders[j+1].getTotalPrice()) {

// Swap orders[j] and orders[j+1]

Order temp = orders[j];

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

orders[j+1] = temp;

}

}

}

}

Quick Sort Implementation:

public static void quickSort(Order[] orders) {

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

}

private 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].getTotalPrice();

int i = low - 1;

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

if (orders[j].getTotalPrice() >= pivot) { // Descending order

i++;

// Swap orders[i] and orders[j]

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;

}

Main Class with Testing:

public class OrderSorting {

public static void main(String[] args) {

// Create sample orders

Order[] orders = {

new Order("O1001", "Alice Johnson", 150.75),

new Order("O1002", "Bob Smith", 89.99),

new Order("O1003", "Charlie Brown", 250.50),

new Order("O1004", "Diana Prince", 45.25),

new Order("O1005", "Ethan Hunt", 175.00)

};

Order[] ordersForBubbleSort = orders.clone();

Order[] ordersForQuickSort = orders.clone();

// Sort with Bubble Sort

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

printOrders(ordersForBubbleSort);

bubbleSort(ordersForBubbleSort);

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

printOrders(ordersForBubbleSort);

// Sort with Quick Sort

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

printOrders(ordersForQuickSort);

quickSort(ordersForQuickSort);

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

printOrders(ordersForQuickSort);

}

private static void printOrders(Order[] orders) {

for (Order order : orders) {

System.out.println(order);

}

}

}

3. Analysis

Performance Comparison:

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) |

Why Quick Sort is Generally Preferred:

1. **Average Case Performance**: O(n log n) vs O(n²) for Bubble Sort
2. **Cache Efficiency**: Quick Sort has better locality of reference
3. **In-Place Sorting**: Uses small additional memory (log n for recursion stack)
4. **Practical Performance**: Typically faster than other O(n log n) algorithms for most cases
5. **Tail Recursion Optimization**: Can be optimized to reduce stack space

When to Use Bubble Sort:

1. **Small Datasets**: For very small arrays (n < 10), Bubble Sort can be competitive
2. **Nearly Sorted Data**: If data is already mostly sorted, Bubble Sort can approach O(n)
3. **Educational Purposes**: Simple to understand and implement

Output:

