**Exercise 3: Sorting Customer Orders**

## **Sorting Algorithms:**

Sorting algorithms arrange data in a specific order (e.g., ascending or descending). Common algorithms include Bubble Sort, Insertion Sort, Quick Sort, and Merge Sort, each with distinct approaches and efficiencies for sorting tasks like ordering customer data by price.

**Bubble Sort:**

**Explanation**: Bubble Sort repeatedly compares and swaps adjacent elements, moving the largest to the end each pass, simple but slow for large lists.

**Java Code Snippet:**

|  |
| --- |
| **public** **class** **BubbleSort** {  **public** **static** **void** **bubbleSort**(**int**[] arr) {  **int** n = arr.length;  **for** (**int** i = **0**; i < n - **1**; i++) {  **for** (**int** j = **0**; j < n - i - **1**; j++) {  **if** (arr[j] > arr[j + **1**]) {  **int** temp = arr[j];  arr[j] = arr[j + **1**];  arr[j + **1**] = temp;  }  }  }  }  } |

Time Complexity: O(n^2) (best, average, worst).

Space Complexity: O(1).

**Insertion Sort:**

**Explanation**: Insertion Sort builds a sorted portion by inserting each element into its correct position, effective for small or nearly sorted data.

**Java Code Snippet:**

|  |
| --- |
| **public** **class** **InsertionSort** {  **public** **static** **void** **insertionSort**(**int**[] arr) {  **int** n = arr.length;  **for** (**int** i = **1**; i < n; i++) {  **int** key = arr[i];  **int** j = i - **1**;  **while** (j >= **0** && arr[j] > key) {  arr[j + **1**] = arr[j];  j--;  }  arr[j + **1**] = key;  }  }  } |

Time Complexity: O(n) (best), O(n^2) (average, worst).

Space Complexity: O(1).

**Quick Sort:**

**Explanation**: Quick Sort selects a pivot, partitions the array into smaller and larger elements, and recursively sorts subarrays, efficient for large datasets.

**Java Code Snippet:**

|  |
| --- |
| **public** **class** **QuickSort** {  **public** **static** **void** **quickSort**(**int**[] arr, **int** low, **int** high) {  **if** (low < high) {  **int** pi = partition(arr, low, high);  quickSort(arr, low, pi - **1**);  quickSort(arr, pi + **1**, high);  }  }  **private** **static** **int** **partition**(**int**[] arr, **int** low, **int** high) {  **int** pivot = arr[high];  **int** i = low - **1**;  **for** (**int** j = low; j < high; j++) {  **if** (arr[j] <= pivot) {  i++;  **int** temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  **int** temp = arr[i + **1**];  arr[i + **1**] = arr[high];  arr[high] = temp;  **return** i + **1**;  }  } |

Time Complexity: O(n log n) (average, best), O(n^2) (worst).

Space Complexity: O(log n).

**Merge Sort:**

**Explanation**: Merge Sort divides the array into halves, recursively sorts them, and merges the sorted halves, stable but requires extra space.

**Java Code Snippet:**

|  |
| --- |
| **public** **class** **MergeSort** {  **public** **static** **void** **mergeSort**(**int**[] arr, **int** l, **int** r) {  **if** (l < r) {  **int** m = l + (r - l) / **2**;  mergeSort(arr, l, m);  mergeSort(arr, m + **1**, r);  merge(arr, l, m, r);  }  }  **private** **static** **void** **merge**(**int**[] arr, **int** l, **int** m, **int** r) {  **int** n1 = m - l + **1**;  **int** n2 = r - m;  **int**[] L = **new** **int**[n1];  **int**[] R = **new** **int**[n2];  **for** (**int** i = **0**; i < n1; i++)  L[i] = arr[l + i];  **for** (**int** j = **0**; j < n2; j++)  R[j] = arr[m + **1** + j];  **int** i = **0**, j = **0**, k = l;  **while** (i < n1 && j < n2) {  **if** (L[i] <= R[j]) {  arr[k] = L[i];  i++;  } **else** {  arr[k] = R[j];  j++;  }  k++;  }  **while** (i < n1) {  arr[k] = L[i];  i++;  k++;  }  **while** (j < n2) {  arr[k] = R[j];  j++;  k++;  }  }  } |

Time Complexity: O(n log n) (best, average, worst).

Space Complexity: O(n).

**Performance Analysis:**

**Time Complexity Comparison**

* **Bubble Sort**: O(n^2) for all cases, due to nested loops for comparisons and swaps.
* **Quick Sort**: O(n log n) average/best, O(n^2) worst (rare, with poor pivots).

**Why Quick Sort is Preferred:**

Quick Sort’s O(n log n) average time complexity outperforms Bubble Sort’s O(n^2) for large datasets, as it reduces comparisons via partitioning. Bubble Sort’s repeated full-array scans are inefficient. Quick Sort sorts in-place, using minimal extra space (O(log n)), and worst-case issues are mitigated with good pivot choices. For sorting large order lists in e-commerce, Quick Sort’s speed is ideal, while Bubble Sort suits only small datasets due to its simplicity.