**Exercise 3: Sorting Customer Orders**

**SCENARIO:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**STEPS:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**UNDERSTAND SORTING ALGORITHMS**

* **Bubble Sort**

Bubble sort is [a sorting algorithm](https://www.programiz.com/dsa/sorting-algorithm) that compares two adjacent elements and swaps them until they are in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

* + **Working of Bubble Sort**

Suppose we are trying to sort the elements in ascending order.

1. **First Iteration (Compare and Swap)**

Starting from the first index, compare the first and the second elements.

If the first element is greater than the second element, they are swapped.

Now, compare the second and the third elements. Swap them if they are not in order.

The above process goes on until the last element. Compare the Adjacent Elements

1. **Remaining Iteration**

The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.

Put the largest element at the end

In each iteration, the comparison takes place up to the last unsorted element.

Compare the adjacent elements

The array is sorted when all the unsorted elements are placed at their correct positions.

The array is sorted if all elements are kept in the right order

* + **Bubble Sort Algorithm**

bubbleSort(array)

for i <- 1 to indexOfLastUnsortedElement-1

if leftElement > rightElement

swap leftElement and rightElement

end bubbleSort

* + **Bubble Sort Complexity:**

|  |  |
| --- | --- |
| Time Complexity |  |
| Best | O(n) |
| Worst | O(n2) |
| Average | O(n2) |
| Space Complexity | O(1) |

* **Insertion Sort Algorithm**

Insertion sort is [a sorting algorithm](https://www.programiz.com/dsa/sorting-algorithm) that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

* + **Working of Insertion Sort:**

Suppose we need to sort the following array.

The first element in the array is assumed to be sorted. Take the second element and store it separately in key.  
  
Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element. If the first element is greater than key, then key is placed in front of the first element.

Now, the first two elements are sorted.  
  
Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array. Place 1 at the beginning

Similarly, place every unsorted element at its correct position. Place 4 behind 1Place 3 behind 1 and the array is sorted

* + **Insertion Sort Algorithm**

insertionSort(array)

mark first element as sorted

for each unsorted element X

'extract' the element X

for j <- lastSortedIndex down to 0

if current element j > X

move sorted element to the right by 1

break loop and insert X here

end insertionSort

* + **Insertion Sort Complexity**

|  |  |
| --- | --- |
| **Time Complexity** |  |
| Best | O(n) |
| Worst | O(n2) |
| Average | O(n2) |
| **Space Complexity** | O(1) |

* **Merge Sort Algorithm:**

Merge Sort is one of the most popular [sorting algorithms](https://www.programiz.com/dsa/sorting-algorithm) that is based on the principle of [Divide and Conquer Algorithm](https://www.programiz.com/dsa/divide-and-conquer).

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

* + **Divide**

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

* + **Conquer**

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

* + **Combine**

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

* + **MergeSort Algorithm:**

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

MergeSort(A, p, r):

if p > r

return

q = (p+r)/2

mergeSort(A, p, q)

mergeSort(A, q+1, r)

merge(A, p, q, r)

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).

The merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.

The merge Step of Merge Sort

Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it, merge step.

The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

* + **Merge Sort Complexity**

|  |  |
| --- | --- |
| Time Complexity |  |
| Best | O(n\*log n) |
| Worst | O(n\*log n) |
| Average | O(n\*log n) |
| Space Complexity | O(n) |

* **Quicksort Algorithm:**

Quicksort is [a sorting algorithm](https://www.programiz.com/dsa/sorting-algorithm) based on the divide and conquer approach where

An array is divided into subarrays by selecting a pivot element (element selected from the array).  
  
While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

* + **Working of Quicksort Algorithm**

1. **Select the Pivot Element**

There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

Select a pivot element

1. **Rearrange the Array**

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

Put all the smaller elements on the left and greater on the right of pivot element

Here's how we rearrange the array:

A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index. Comparison of pivot element with element beginning from the first index

If the element is greater than the pivot element, a second pointer is set for that element. If the element is greater than the pivot element, a second pointer is set for that element.

Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier. Pivot is compared with other elements.

Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element. The process is repeated to set the next greater element as the second pointer.

The process goes on until the second last element is reached. The process goes on until the second last element is reached.

Finally, the pivot element is swapped with the second pointer. Finally, the pivot element is swapped with the second pointer.

1. **Divide Subarrays**

Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated. Select pivot element of in each half and put at correct place using recursion

The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

* + **Quick Sort Algorithm**

quickSort(array, leftmostIndex, rightmostIndex)

if (leftmostIndex < rightmostIndex)

pivotIndex <- partition(array,leftmostIndex, rightmostIndex)

quickSort(array, leftmostIndex, pivotIndex - 1)

quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)

set rightmostIndex as pivotIndex

storeIndex <- leftmostIndex - 1

for i <- leftmostIndex + 1 to rightmostIndex

if element[i] < pivotElement

swap element[i] and element[storeIndex]

storeIndex++

swap pivotElement and element[storeIndex+1]

return storeIndex + 1

* + **Quicksort Complexity**

|  |  |
| --- | --- |
| Time Complexity |  |
| Best | O(n\*log n) |
| Worst | O(n2) |
| Average | O(n\*log n) |
| Space Complexity | O(log n) |

**SETUP AND IMPLEMENTATION:**

package cognizantJavaFSE;

public 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;

}

public String getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

@Override

public String toString() {

return "Order{" +

"orderId='" + orderId + '\'' +

", customerName='" + customerName + '\'' +

", totalPrice=" + totalPrice +

'}';

}

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()) {

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].getTotalPrice();

int i = (low - 1);

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

if (orders[j].getTotalPrice() < 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", 250.0),

new Order("2", "Bob", 150.0),

new Order("3", "Charlie", 200.0),

};

System.out.println("Original Order Array:");

for (Order order : orders) {

System.out.println(order);

}

Order[] bubbleSortedOrders = orders.clone();

bubbleSort(bubbleSortedOrders);

System.out.println("\nBubble Sorted Order Array:");

for (Order order : bubbleSortedOrders) {

System.out.println(order);

}

Order[] quickSortedOrders = orders.clone();

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

System.out.println("\nQuick Sorted Order Array:");

for (Order order : quickSortedOrders) {

System.out.println(order);

}

}

}

**PERFORMANCE ANALYSIS**

* Time Complexity Comparison
* **Bubble Sort:**
  + **Best Case:** O(n) - Occurs when the array is already sorted.
  + **Average Case:** O(n^2) - Involves comparing and swapping elements in a nested loop.
  + **Worst Case:** O(n^2)- Occurs when the array is sorted in reverse order.
* **Quick Sort:**
  + **Best Case:** O(nlogn) - Occurs when the pivot divides the array into two nearly equal halves.
  + **Average Case:** O(nlogn) - Generally, the pivot will partition the array reasonably well.
  + **Worst Case:** O(n^2) - Occurs when the pivot is the smallest or largest element, resulting in highly unbalanced partitions. This can be mitigated by using techniques like choosing a random pivot or the median-of-three.

#### Why Quick Sort is Generally Preferred Over Bubble Sort

1. **Efficiency on Large Datasets:**
   * Quick Sort has an average time complexity of O(nlogn), making it significantly faster than Bubble Sort's O(n^2)for large datasets. The logarithmic factor makes a considerable difference in execution time as the input size increases.
2. **Divide and Conquer Strategy:**
   * Quick Sort's divide-and-conquer strategy is efficient at breaking down the problem into smaller subproblems, leading to faster sorting overall. This makes it adaptable to various types of input data.
3. **Fewer Comparisons and Swaps:**
   * Quick Sort generally performs fewer comparisons and swaps than Bubble Sort. Bubble Sort repeatedly compares and swaps adjacent elements, leading to a higher number of operations, especially in large or reversed arrays.
4. **Better Performance on Average:**
   * Despite its worst-case time complexity of O(n^2), Quick Sort's average performance is much better due to its balanced partitioning approach in most cases. Using strategies like random pivot selection or median-of-three can help avoid the worst-case scenario.
5. **In-Place Sorting:**
   * Quick Sort is an in-place sort (requiring only a small, constant amount of extra storage space), which is beneficial for memory efficiency. While Bubble Sort is also in-place, its performance drawback outweighs this benefit.