**Selection Sort**

Here's a detailed explanation of each part of the code:

**Array Initialization:** The array a is initialized with the elements that need to be sorted.

Variable Declarations: min is used to store the index of the minimum element, and temp is used for swapping elements.

**Outer Loop:** The outer loop iterates through each element in the array.

Assumption of Minimum: At the start of each outer loop iteration, the current position i is assumed to be the minimum.

**Inner Loop:** The inner loop starts from i + 1 and searches for the minimum element in the remaining unsorted portion of the array.

**Finding the Minimum:** If an element smaller than the current minimum is found, min is updated to the index of this new minimum.

**Swapping:** After the inner loop completes, the minimum element found is swapped with the element at the current position i.

**Print Sorted Array:** After sorting is complete, the array is printed in sorted order.

This code correctly implements the selection sort algorithm and prints the sorted array at the end.

**Insertion Sort**

Detailed Explanation:

**1 Array Initialization:** The array a is initialized with the elements that need to be sorted.

**2 Variable Declarations:** temp is used to store the current element being compared, and j is used for the inner loop index.

**3 Outer Loop:** The outer loop starts from the second element (index 1) and iterates through the array.

Store Current Element: The current element a[i] is stored in temp.

**5 Initialize j:** j is initialized to the current index i.

**6 Inner Loop:** The inner loop shifts elements of the sorted portion of the array to the right to create space for the current element.

* The condition j > 0 && a[j-1] > temp ensures that elements are shifted only if they are greater than temp and j is greater than 0.
* Elements are shifted to the right by assigning a[j-1] to a[j].
* j is decremented to continue checking the previous elements.

**7 Place Element:** The current element temp is placed in its correct position at a[j].

**8 Print Sorted Array:** After sorting is complete, the array is printed in sorted order.

This code correctly implements the insertion sort algorithm and prints the sorted array at the end.

**Merge Sort**

Detailed Explanation:

**1 Class Variables:**

* array: The main array to be sorted.
* tempMergeArr: Temporary array used for merging the sorted subarrays.
* length: Length of the main array.

**2 Main Method:**

* Initializes the array inputArr to be sorted.
* Creates an instance of MergeSort and calls the sort method.
* Prints the sorted array.

**3 Sort Method:**

* Sets the class variables array, length, and tempMergeArr.
* Calls divideArray to start the recursive division of the array.

**4 DivideArray Method:**

* Recursively divides the array into two halves until each subarray contains one element.
* Calls itself to sort the left and right halves.
* Calls mergeArray to merge the sorted subarrays.

**5 MergeArray Method:**

* Copies the current subarray into the temporary array tempMergeArr.
* Initializes indices for the left subarray (i), right subarray (j), and the main array (k).
* Merges the elements from the temporary array back into the main array in sorted order.
* Copies any remaining elements from the left subarray into the main array (elements in the right subarray are already in place).

**Quick Sort**

**1 Main Method:**

* Initializes the array arr to be sorted.
* Creates an instance of QuickSort.
* Calls quickSortRecursion to sort the array.
* Calls printArray to print the sorted array.

**2 Partition Method:**

* Chooses the pivot element (middle element of the array).
* Uses two pointers (low and high) to find elements that need to be swapped.
* Swaps elements to ensure that elements less than the pivot are on the left and elements greater than the pivot are on the right.
* Continues until the low index is greater than the high index.
* Returns the partition index (low).

**3 QuickSortRecursion Method:**

* Calls the partition method to get the partition index.
* Recursively sorts the subarray to the left of the partition index.
* Recursively sorts the subarray to the right of the partition index.

**4 PrintArray Method:**

* Prints the elements of the array.

**Fixes:**

1. Return Type: The partition method should return an int (the partition index).
2. Method Calls: Corrected method calls in quickSortRecursion to ensure proper sorting.
3. Instance Creation: Created an instance of QuickSort in the main method to call the instance methods.

**Heap Sort**

**1 Main Method:**

* Initializes the array arr to be sorted.
* Creates an instance of HeapSort.
* Calls sort to perform heap sort on the array.
* Prints the sorted array.

**2 Sort Method:**

* Determines the length of the array leng.
* Builds a max heap from the array:
  + The loop starts from the last non-leaf node and calls heapify to ensure that the subtree rooted at each node is a max heap.
* Extracts elements from the heap one by one:
  + The loop swaps the root of the heap (largest element) with the last element of the heap and reduces the heap size.
  + Calls heapify on the reduced heap to restore the heap property.

**3 Heapify Method:**

* Ensures that the subtree rooted at index i is a max heap.
* Initializes largest as the root.
* Calculates the indices of the left (li) and right (ri) children.
* Compares the root with its left and right children and updates largest if one of the children is larger.
* If largest is not the root, swaps the root with the largest element and recursively calls heapify on the affected subtree.

This implementation ensures that the array is sorted using the heap sort algorithm and prints the sorted array at the end.