Task1:

* Consider an array of size n.
* Search for largest number from arr[0] to arr[n-1].
* Now swap the element arr[n-1] with the largest number and decrement length of array as n--.
* Now repeat the process till only the first element is left and we have a sorted array using selection sort.

Task2:

Function SelectionSort(Array A, Integer n)

// Outer loop: Iterate from the last element down to the second element.

// 'i' marks the current "end" of the unsorted portion of the array.

For i from n-1 down to 1:

// Assume the element at index 0 is the current maximum within A[0...i].

Set maxIndex to 0

// Inner loop: Find the index of the true maximum element in the unsorted subarray A[0...i].

For j from 1 up to i:

// If the element at current 'j' is greater than or equal to the element at 'maxIndex'

If A[j] >= A[maxIndex]:

// Update maxIndex to 'j' as we found a new maximum

Set maxIndex to j

// After the inner loop, maxIndex holds the index of the largest element

// in the subarray A[0...i].

// Swap the element at the current "end" of the unsorted portion (A[i])

// with the largest element found (A[maxIndex]).

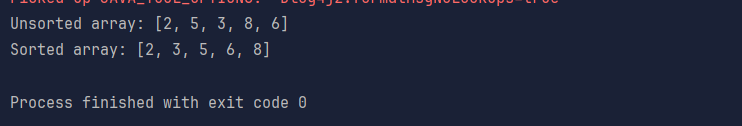
Swap A[i] with A[maxIndex]

// End of outer loop

End Function

Task3:

import **java.util.Arrays**;  
  
public class **Task3** {  
 static void selectionSort(int[] arr){  
 int n= arr.length;  
 for (int i=n-1;i>=1;i--){  
 int maxIndex=0;  
 for (int j=1; j<=i;j++){  
 if (arr[j]>=arr[maxIndex]){  
 maxIndex=j;  
 }  
 }  
 int temp = arr[i];  
 arr[i] = arr[maxIndex];  
 arr[maxIndex] = temp;  
 }  
 }  
  
 public static void main(**String**[] args) {  
 int[] nums = {2, 5, 3, 8, 6};  
 **System**.*out*.println("Unsorted array: "+ **Arrays**.*toString*(nums));  
 *selectionSort*(nums);  
 **System**.*out*.println("Sorted array: "+ **Arrays**.*toString*(nums));  
  
 }  
}



Task4:

* Starting at the first element, compare the pair of first two elements, if arr[n-1]>arr[n], swap the two elements.
* If arr[n]>arr[n-1], move to the pair of arr[n] and arr[n+1].
* Keep going till the end of the array.

Task5:

Function BubbleSort(Array A, Integer n)

// Outer loop: Iterate from the end of the array downwards.

// 'i' represents the boundary between the sorted (right) and unsorted (left) parts.

// In each pass, the largest unsorted element "bubbles" up to position 'i'.

For i from n-1 down to 1:

// Inner loop: Traverse the unsorted portion of the array.

// Compare adjacent elements and swap them if they are in the wrong order.

For j from 1 up to i:

// If the current element (A[j-1]) is greater than the next element (A[j])

If A[j-1] > A[j]:

// Swap them to put the larger element to the right

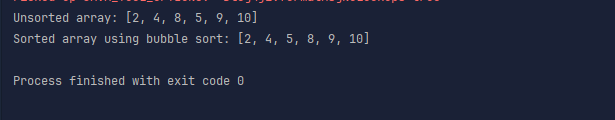
Swap A[j] with A[j-1]

// End of outer loop

End Function

Task6:

import **java.util.Arrays**;  
  
public class **Task6** {  
 static void bubbleSort(int[] arr){  
 int n=arr.length;  
 for (int i=n-1;i>=1;i--){  
 for (int j=1; j<=i;j++){  
 if (arr[j-1]>arr[j]){  
 int temp = arr[j];  
 arr[j] = arr[j-1];  
 arr[j-1] = temp;  
 }  
 }  
 }  
 }  
  
 public static void main(**String**[] args) {  
 int[] nums = {2, 4, 8, 5, 9, 10};  
 **System**.*out*.println("Unsorted array: "+**Arrays**.*toString*(nums));  
 *bubbleSort*(nums);  
 **System**.*out*.println("Sorted array using bubble sort: "+ **Arrays**.*toString*(nums));  
 }  
}



Task7:

Algo for insertion sort:

* First element that is 0th element is already considered sorted in insertion sort.
* While inserting the second element, compare it with the elements present, and place it accordingly.
* Similarly, for the next element, compare it with the current elements, and place it either at the beginning, end or in the middle, whichever is correct order. Inserting in right will increase the length of the array, inserting in middle, will increase the length of array and shift the elements on right by one place to the right.
* Inserting at the beginning will increase the size of array by 1 and shift all the elements to the right by 1 position.
* Keep going till all elements are inserted in a sorted order.

Task8:

Function InsertionSort(Array A, Integer n)

// Outer loop: Iterate from the second element (index 1) to the last element.

// 'i' marks the element currently being considered for insertion.

For i from 1 to n-1:

// Store the current element as 'key'. This is the element to be inserted.

Set key = A[i]

// Inner loop: Move elements of the sorted subarray A[0...i-1] that are

// greater than 'key' to one position ahead of their current position.

Set j = i - 1

While j >= 0 AND A[j] > key:

Set A[j+1] = A[j] // Shift element to the right

Decrement j // Move to the left in the sorted subarray

// Place 'key' into its correct position in the sorted subarray.

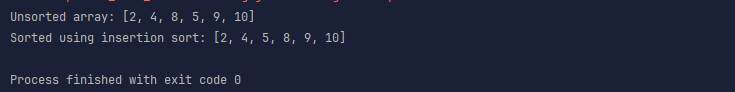
Set A[j+1] = key

// End of outer loop

End Function

Task9:

import **java.util.Arrays**;  
  
public class **Task9** {  
 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;  
 }  
 }  
  
 public static void main(**String**[] args) {  
 int[] nums = {2, 4, 8, 5, 9, 10};  
 **System**.*out*.println("Unsorted array: " + **Arrays**.*toString*(nums));  
 *insertionSort*(nums);  
 **System**.*out*.println("Sorted using insertion sort: " + **Arrays**.*toString*(nums));  
 }  
}



Task10:

**Advantages of Bubble Sort:**

1. Simplicity (Easy to understand and implement)
2. Stability
3. In-Place sorting (O(1) space complexity)
4. Relatively efficient for very small or nearly sorted data sets

**Disadvantages of Bubble Sort (Poor Performance - Limitations):**

1. Poor Time Complexity (O(n2) in worst and average cases)
2. Excessive number of swaps
3. Inefficient for large data sets
4. Not adaptive (without optimization)

Task11:

public class RecLoop {

     public int calc(int n) {

        if (n == 0) return 0;

        return n + calc**(n-1);**

    }

Task 12:

Algo for Merge sort:

* Divide the initial array into two halves.
* Further divide each half till only one element is left per part.
* Now start merging by taking the divided elements as pairs and comparing them.
* Further merge these two element parts by comparing their first elements and placing them into a new array and keep going until a fully sorted array is left.

Task13:

Pseudo code for merge sort:

Function MergeSort(Array A)

// Base Case: If the array has 0 or 1 element, it's already sorted.

If length of A <= 1:

Return A // It's already sorted!

// 1. Divide: Split the array into two halves.

Set mid = length of A / 2

Set LeftHalf = A from index 0 to mid-1

Set RightHalf = A from index mid to end

// 2. Conquer (Recursively Sort): Sort each half.

Set SortedLeftHalf = MergeSort(LeftHalf)

Set SortedRightHalf = MergeSort(RightHalf)

// 3. Combine: Merge the two sorted halves back together.

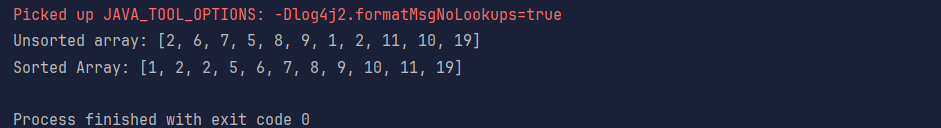
Set ResultArray = Merge(SortedLeftHalf, SortedRightHalf)

Return ResultArray

End Function

Task14:

import **java.util.Arrays**;  
  
public class **Task14** {  
  
 static void mergeSort(int[] arr, int left, int right) {  
 if (left < right) {  
 int mid = left + (right - left) / 2;  
  
 *mergeSort*(arr, left, mid);  
 *mergeSort*(arr, mid + 1, right);  
  
 *merge*(arr, left, mid, right);  
 }  
 }  
  
 static void merge(int[] arr, int left, int mid, int right) {  
 int n1 = mid - left + 1;  
 int n2 = right - mid;  
  
 int[] L = new int[n1];  
 int[] R = new int[n2];  
  
 for (int i = 0; i < n1; ++i) {  
 L[i] = arr[left + i];  
 }  
 for (int j = 0; j < n2; ++j) {  
 R[j] = arr[mid + 1 + j];  
 }  
  
 int i = 0;  
 int j = 0;  
 int k = left;  
  
 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++;  
 }  
 }  
  
 public static void main(**String**[] args) {  
 int[] nums = {2, 6, 7, 5, 8, 9, 1, 2, 11, 10, 19};  
 **System**.*out*.println("Unsorted array: " + **Arrays**.*toString*(nums));  
 *mergeSort*(nums, 0, nums.length-1);  
 **System**.*out*.println("Sorted Array: "+ **Arrays**.*toString*(nums));  
 }  
}



Task15:

* Pick an element (random known element) as a pivot element from the given array.
* Now go through the other elements, if smaller than pivot element, place it in smaller pile. If larger than pivot element, place it in larger pile, if same size as your pivot element, it can be placed in any one of the piles.
* Now for the ‘smaller’ pile, repeat the process of taking a pivot element and dividing the elements.
* Keep going till you only have one element per pile.
* A single element is always sorted.
* There is no need to combine, since we’ve already sorted while splitting, merging the elements will make it so that all the elements fall into the place as a sorted array.
* Now the smaller sorted group is placed at the left of the pivot element and the larger group is placed at the right of the pivot.

Task16:

Function QuickSort(Array A, Integer low, Integer high)

* Base Case: If the subarray has 0 or 1 element, it's already sorted.
* This means 'low' is not less than 'high'.

If low < high:

* 1. Partition: Choose a 'pivot' element and rearrange the subarray

so that elements smaller than the pivot are to its left,

and elements larger than the pivot are to its right.

'pivotIndex' will be the final sorted position of the pivot.

* Set pivotIndex = Partition(A, low, high)
* 2. Conquer (Recursively Sort):
* Recursively sort the subarray of elements SMALLER than the pivot.

QuickSort(A, low, pivotIndex - 1)

* Recursively sort the subarray of elements LARGER than the pivot.

QuickSort(A, pivotIndex + 1, high)

End Function

Task17:

import **java.util.Arrays**;  
  
public class **Task17** {  
  
 static void quickSort(int[] arr, int low, int high) {  
 if (low < high) {  
 int pivotIndex = *partition*(arr, low, high);  
  
 *quickSort*(arr, low, pivotIndex - 1);  
 *quickSort*(arr, pivotIndex + 1, high);  
 }  
 }  
  
 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;  
 }  
  
 public static void main(**String**[] args) {  
 int[] nums = {2, 6, 7, 5, 8, 9, 1, 2, 11, 10, 19};  
 **System**.*out*.println("Unsorted array: " + **Arrays**.*toString*(nums));  
 *quickSort*(nums, 0, nums.length-1);  
 **System**.*out*.println("SOrted using quick sort: "+**Arrays**.*toString*(nums) );  
 }  
}

