Task 👍1:

Write an algorithm / steps for selection sort.

1. Start

2. For i = 0 to N-1 (where N is the size of the array):

a. Set minIndex = i

b. For j = i+1 to N-1:

i. If array[j] < array[minIndex], then set minIndex = j

c. If minIndex != i, then:

i. Swap array[i] and array[minIndex]

3. End

Task 2:

Write a pseudo code for the selection sort

SelectionSort(array)

n = length of array

// Traverse through all elements of the array

for i = 0 to n-1 do

// Assume the minimum element is the first unsorted element

minIndex = i

// Find the minimum element in the remaining unsorted part

for j = i+1 to n-1 do

if array[j] < array[minIndex] then

minIndex = j

end if

end for

// Swap the found minimum element with the first unsorted element

if minIndex != i then

swap(array[i], array[minIndex])

end if

end for

end SelectionSort

Task 3:

Wap to make sure your list is sorted using selection sort.

public class SelectionSort {

// Function to perform Selection Sort

public static void selectionSort(int[] arr) {

int n = arr.length;

// Traverse through all elements of the array

for (int i = 0; i < n - 1; i++) {

// Assume the minimum element is the current element

int minIndex = i;

// Find the minimum element in the unsorted portion of the array

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap the found minimum element with the first unsorted element

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

// Function to verify if the array is sorted

public static boolean isSorted(int[] arr) {

for (int i = 1; i < arr.length; i++) {

if (arr[i] < arr[i - 1]) {

return false; // If any element is smaller than the previous one, it's not sorted

}

}

return true; // The array is sorted

}

// Main method to test the program

public static void main(String[] args) {

// Example array

int[] arr = {64, 25, 12, 22, 11};

// Print the original array

System.out.println("Original array:");

printArray(arr);

// Sort the array using Selection Sort

selectionSort(arr);

// Print the sorted array

System.out.println("Sorted array:");

printArray(arr);

// Verify if the array is sorted

if (isSorted(arr)) {

System.out.println("The list is sorted correctly.");

} else {

System.out.println("The list is NOT sorted correctly.");

}

}

// Helper function to print the array

public static void printArray(int[] arr) {

for (int i : arr) {

System.out.print(i + " ");

}

System.out.println();

}

}

Bubble Sort:

Task 4:

Write algorithm for the Bubble sort.

 **Start** with the entire list (array).

 **Compare adjacent elements** in the array.

* For each pair of adjacent elements, if the first element is larger than the second, **swap them**.

 After each pass through the list, the largest element will be "bubbled" to the end of the array.

 Repeat the process for the remaining unsorted portion of the array (ignoring the last sorted element after each pass).

 Continue this until no swaps are needed, which indicates the array is sorted.

Task 5:

Write pseudo code for the bubble sort

BubbleSort(array)

n = length of array

// Outer loop for each pass through the array

for i = 0 to n-1 do

swapped = false // Flag to track if a swap was made

// Inner loop to compare adjacent elements

for j = 0 to n-i-2 do

if array[j] > array[j+1] then

// Swap if elements are in the wrong order

swap(array[j], array[j+1])

swapped = true

end if

end for

// If no swaps were made, the array is already sorted

if not swapped then

break // Exit early since the array is sorted

end if

end for

end BubbleSort

Task 6:

Wap to make sure your list is sorted using Bubble sort.

public class BubbleSort {

// Function to perform Bubble Sort

public static void bubbleSort(int[] arr) {

int n = arr.length;

// Outer loop for each pass

for (int i = 0; i < n - 1; i++) {

boolean swapped = false; // Flag to check if any swap occurred

// Inner loop to compare adjacent elements

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap the elements if they are in the wrong order

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true; // A swap occurred

}

}

// If no elements were swapped, the array is already sorted

if (!swapped) {

break; // Early exit if no swaps were made

}

}

}

// Function to verify if the array is sorted

public static boolean isSorted(int[] arr) {

for (int i = 1; i < arr.length; i++) {

if (arr[i] < arr[i - 1]) {

return false; // If the current element is smaller than the previous one, it's not sorted

}

}

return true; // The array is sorted

}

// Helper function to print the array

public static void printArray(int[] arr) {

for (int i : arr) {

System.out.print(i + " ");

}

System.out.println();

}

public static void main(String[] args) {

// Example array

int[] arr = {64, 25, 12, 22, 11};

// Print the original array

System.out.println("Original array:");

printArray(arr);

// Sort the array using Bubble Sort

bubbleSort(arr);

// Print the sorted array

System.out.println("Sorted array:");

printArray(arr);

// Verify if the array is sorted

if (isSorted(arr)) {

System.out.println("The list is sorted correctly.");

} else {

System.out.println("The list is NOT sorted correctly.");

}

}

}

Insertion sort:

Task 7:

Write an algorithm for the Insertion sort

 **Start** with the second element of the array (index 1), because the first element is considered already sorted.

 **Compare** the current element (let's call it key) with the element just before it.

 If the key is smaller than the previous element, **shift** the previous element to the right to make space for the key.

 Continue this process of shifting until you find the correct position for the key.

 **Insert** the key into its correct position.

 Repeat the process for all elements in the array (from the second element to the last).

 Once you finish, the array is sorted.

Task 8:

Write pseudocode for the Insertion sort

InsertionSort(A)

for i = 1 to length(A) - 1:

key = A[i] // the element to be inserted

j = i - 1 // index of the element before `key`

// Shift elements of A[0..i-1] that are greater than key, to one position ahead of their current position

while j >= 0 and A[j] > key:

A[j + 1] = A[j]

j = j - 1

A[j + 1] = key // place the key in its correct position

Task 9:

Wap to make sure your list is sorted using Insertion sort.

import java.util.Arrays;

public class InsertionSort {

// Insertion Sort Function

public static void insertionSort(int[] arr) {

for (int i = 1; i < arr.length; i++) {

int key = arr[i]; // Element to be inserted in the sorted part of the array

int j = i - 1; // Index of the element just before the key

// Move elements of arr[0..i-1] that are greater than key to one position ahead

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j]; // Shift element to the right

j--; // Move leftwards

}

// Place the key in the correct position

arr[j + 1] = key;

}

}

public static void main(String[] args) {

// Example list to be sorted

int[] myList = {12, 11, 13, 5, 6};

// Print the original list

System.out.println("Original List: " + Arrays.toString(myList));

// Sort the list using Insertion Sort

insertionSort(myList);

// Print the sorted list

System.out.println("Sorted List: " + Arrays.toString(myList));

}

}

Output:

Original List: [12, 11, 13, 5, 6]

Sorted List: [5, 6, 11, 12, 13]

Task 10:

What are the advantages and disadvantages of Bubble sort Algo?

List them

note:

Poor performance - limitations of bubble sort

**Advantages of Bubble Sort:**

1. **Simplicity**:
   * Bubble Sort is one of the simplest sorting algorithms to understand and implement. It only requires basic loops and comparisons, making it beginner-friendly for anyone learning sorting algorithms.
2. **In-place Sorting**:
   * It sorts the list in place, meaning it doesn't require extra memory (apart from the input list itself). This makes it memory-efficient compared to other sorting algorithms like Merge Sort, which requires additional space for recursion.
3. **Stable Sorting**:
   * Bubble Sort is a **stable** sorting algorithm. If two elements have the same value, they maintain their relative order after sorting. This is important when dealing with complex data structures where the order of equal elements matters.
4. **Early Termination (Optimized Version)**:
   * If the list is already sorted (or partially sorted), Bubble Sort can terminate early. By adding a flag to track whether any swaps occurred in a pass, the algorithm can stop if no swaps are made in a full iteration, which improves its performance in best-case scenarios.

**Disadvantages of Bubble Sort:**

1. **Poor Time Complexity**:
   * The biggest limitation of Bubble Sort is its **time complexity**. In the worst and average case, Bubble Sort has an **O(n²)** time complexity, where **n** is the number of elements in the list. This makes it very inefficient for large datasets.
     + **Worst Case:** O(n²) (when the list is in reverse order)
     + **Best Case:** O(n) (when the list is already sorted, with optimization)
     + **Average Case:** O(n²)
2. **Inefficient for Large Data**:
   * Due to its quadratic time complexity, Bubble Sort becomes impractical for large data sets. It performs significantly worse than more efficient algorithms like Merge Sort, Quick Sort, or even Insertion Sort for larger arrays.
3. **Excessive Comparisons**:
   * Bubble Sort performs a lot of redundant comparisons, especially when the list is nearly sorted. In each pass, the algorithm compares adjacent elements, even though many comparisons might not be necessary, resulting in excessive computational overhead.
4. **Not Adaptive (in its original form)**:
   * The original Bubble Sort algorithm does not take advantage of already partially sorted lists well. Although an optimized version of Bubble Sort exists that terminates early when no swaps occur, the basic version of the algorithm does not adjust based on the data's structure.
5. **Limited Practical Use**:
   * Due to its inefficiency for larger lists, Bubble Sort is generally not used in practical applications where performance is a concern. It is more of an educational algorithm to demonstrate basic sorting principles rather than a tool for production-level sorting tasks.
6. **Swaps Are Expensive**:
   * Although Bubble Sort swaps adjacent elements, frequent swapping can be inefficient, especially for large datasets. While swapping adjacent elements doesn't require additional memory, it still involves additional operations (comparisons and assignments).

**Summary of Advantages and Disadvantages of Bubble Sort:**

**Advantages:**

* Simple to understand and implement
* In-place sorting (no extra memory required)
* Stable (keeps the relative order of equal elements)
* Can terminate early if the list is already sorted (with optimization)

**Disadvantages:**

* Poor performance for large lists (O(n²) time complexity)
* Inefficient for large datasets
* Excessive unnecessary comparisons
* Not adaptive in its original form (doesn’t take advantage of partially sorted lists)
* Limited real-world use due to inefficiency
* Frequent swaps can be costly in terms of performance

Task 11:

This code is going overflow of stack.. Can you plz help me fix it guys.. ☹️

Note:

Plz be careful: Because recursive calls consume stack memory for every invocation and excessive depth can exceed system limits also..

public class RecLoop {

     public int calc(int n) {

        if (n == 0) return 0;

        return n + calc(n);

    }

public class RecLoop {

public int calc(int n) {

// Base case: when n is 0, stop recursion

if (n == 0) {

return 0;

}

// Recursive case: reduce n by 1 to eventually reach the base case

return n + calc(n - 1); // Decrease n with each recursive call

}

public static void main(String[] args) {

RecLoop recLoop = new RecLoop();

int result = recLoop.calc(5); // Example input

System.out.println("Result: " + result); // Should output 15 (5+4+3+2+1+0)

}

}

Output:

calc(5) -> 5 + calc(4)

calc(4) -> 4 + calc(3)

calc(3) -> 3 + calc(2)

calc(2) -> 2 + calc(1)

calc(1) -> 1 + calc(0)

calc(0) -> 0

TAsk 12:

Algo for merge sort,

#### ****Steps in Merge Sort****:

1. **Divide**: Recursively divide the list into two halves.
2. **Conquer**: Sort each half.
3. **Combine**: Merge the two halves back together in sorted order.

Task 13

pseudo code for merge sort,

 **MergeSort(arr)**:

* If the length of arr is greater than 1:
  1. Find the middle point of the array: middle = len(arr) // 2.
  2. Split the list into two halves:
     + left\_half = arr[:middle]
     + right\_half = arr[middle:]
  3. Recursively sort the left half: MergeSort(left\_half)
  4. Recursively sort the right half: MergeSort(right\_half)
  5. Merge the sorted halves back together using Merge(left\_half, right\_half).

 **Merge(left, right)**:

* Initialize pointers i, j, and k for left, right, and the merged array respectively.
* Compare the elements from both arrays (left and right):
  + If left[i] is smaller, place it in the merged array, and move the pointer i forward.
  + If right[j] is smaller, place it in the merged array, and move the pointer j forward.
* Once one of the arrays is exhausted, copy the remaining elements from the other array into the merged array.

TSK 14

code for Merge sort

import java.util.Arrays;

public class MergeSort {

// Main function that implements Merge Sort

public static void mergeSort(int[] arr) {

if (arr.length > 1) {

// Find the middle point of the array

int mid = arr.length / 2;

// Split the array into two halves

int[] leftHalf = Arrays.copyOfRange(arr, 0, mid);

int[] rightHalf = Arrays.copyOfRange(arr, mid, arr.length);

// Recursively sort the two halves

mergeSort(leftHalf);

mergeSort(rightHalf);

// Merge the sorted halves

merge(arr, leftHalf, rightHalf);

}

}

// Merge two sorted halves into a single sorted array

private static void merge(int[] arr, int[] left, int[] right) {

int i = 0, j = 0, k = 0;

// Compare and merge the two halves

while (i < left.length && j < right.length) {

if (left[i] <= right[j]) {

arr[k] = left[i];

i++;

} else {

arr[k] = right[j];

j++;

}

k++;

}

// If there are remaining elements in the left half, copy them

while (i < left.length) {

arr[k] = left[i];

i++;

k++;

}

// If there are remaining elements in the right half, copy them

while (j < right.length) {

arr[k] = right[j];

j++;

k++;

}

}

// Main method to test the merge sort

public static void main(String[] args) {

int[] arr = {38, 27, 43, 3, 9, 82, 10};

System.out.println("Original Array: " + Arrays.toString(arr));

mergeSort(arr);

System.out.println("Sorted Array: " + Arrays.toString(arr));

}

}

Task 15:

Algo fro quick sort

 **Choose a Pivot**: Select an element from the array (typically the last element or the first element, but you can choose other strategies).

 **Partitioning**: Rearrange the array so that elements smaller than the pivot come before it and elements greater than the pivot come after it.

 **Recursively sort** the sub-arrays formed by dividing the array at the pivot.

Task 16:

Pseudo code for quick sort

 **QuickSort(arr, low, high)**:

* If low < high:
  + Choose a pivot index (typically high).
  + Partition the array and get the pivot index pi.
  + Recursively sort the sub-arrays:
    - QuickSort(arr, low, pi - 1) (left side)
    - QuickSort(arr, pi + 1, high) (right side)

 **Partition(arr, low, high)**:

* Choose a pivot element.
* Initialize i = low - 1.
* Iterate j from low to high - 1:
  + If arr[j] <= pivot, increment i, and swap arr[i] and arr[j].
* Swap arr[i + 1] with arr[high] to place the pivot in the correct position.
* Return i + 1

Task 17:

Code for Quick sort

public class QuickSort {

// Main function to perform quick sort

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

// Partition the array and get the pivot index

int pi = partition(arr, low, high);

// Recursively sort the sub-arrays

quickSort(arr, low, pi - 1); // Sort the left side

quickSort(arr, pi + 1, high); // Sort the right side

}

}

// Partition function to place the pivot in the correct position

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high]; // Pivot is chosen as the last element

int i = low - 1; // Index for the smaller element

// Traverse through the array

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

// If the current element is smaller or equal to the pivot

if (arr[j] <= pivot) {

i++; // Move the boundary for smaller elements

// Swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap the pivot element with the element at arr[i + 1]

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1; // Return the pivot index

}

// Main method to test the quick sort

public static void main(String[] args) {

int[] arr = {38, 27, 43, 3, 9, 82, 10};

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

printArray(arr);

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

System.out.println("Sorted Array: ");

printArray(arr);

}

// Helper method to print the array

private static void printArray(int[] arr) {

for (int i : arr) {

System.out.print(i + " ");

}

System.out.println();

}

}