**DAY16 – July 11th**

**TASK1 - SELECTION SORT**

**Algorithm**

**Step 1 -** For i to n-1

Set maxIndex = 0 or i // We will assume the last element as Max number.

Repeat for i to n-1 down to 1.

**Step 2 –** For j from 1 to i

Checked if arr[j] > arr[maxIndex] // checking for the largest number.

maxIndex = j; //found largest number / updated

**Step 3 –** If found ,

Swap (arr[i], arr[maxIndex])

Array is now sorted in descending order.

**TASK 2 – Pseudocode for Selection sort**

Function selectionSortDescending(arr, n)

for i ← n - 1 down to 1

maxIndex ← 0

for j ← 1 to i

if arr[j] > arr[maxIndex] then

maxIndex ← j

end if

end for

// Swap arr[maxIndex] with arr[i]

temp ← arr[maxIndex]

arr[maxIndex] ← arr[i]

arr[i] ← temp

end for

end Function

**TASK 3 – Code for Selection Sort**

package DSA;  
  
class SelectionSort {  
 public static void selectionSort(int[] arr) {  
 int n = arr.length;  
  
 // Move from end to start, placing the largest element at position i  
 for (int i = n - 1; i >= 1; i--) {  
 int maxIndex = 0;  
  
 // Find the index of the largest element in [0 to i]  
 for (int j = 1; j <= i; j++) {  
 if (arr[j] > arr[maxIndex]) {  
 maxIndex = j;  
 }  
 }  
  
 // Swap the largest element with the element at position i  
 int temp = arr[maxIndex];  
 arr[maxIndex] = arr[i];  
 arr[i] = temp;  
 }  
 }  
  
 // Utility method to print the array  
 public static void printArray(int[] arr) {  
 for (int num : arr) {  
 System.*out*.print(num + " ");  
 }  
 System.*out*.println();  
 }  
  
 // Test the sort  
 public static void main(String[] args) {  
 int[] arr = {64, 25, 12, 22, 11};  
 System.*out*.println("Original array:");  
 *printArray*(arr);  
  
 *selectionSort*(arr);  
  
 System.*out*.println("Sorted array:");  
 *printArray*(arr);  
 }  
}

**Output**

Original array:

64 25 12 22 11

Sorted array:

11 12 22 25 64

**BUBBLE SORT**

**TASK 4 – Algorithm for Bubble sort**

* Repeat the following steps for i = 0 to n – 1 : This outer loop handles how many passes are made
* For each i, repeat the following steps for j = 0 to n - i - 1: This inner loop compares adjacent elements
* If arr[j] > arr[j + 1], then: - Check if the first element in the input array is greater than the next element in the array.
* Swap arr[j] and arr[j + 1] - If it is greater, swap the two elements; otherwise move the pointer forward in the array.
* Repeat steps 3–4 until the array is sorted.
* Print the sorted Array.

**TASK 5 – PseudoCode for Bubble sort**

procedure BubbleSort(arr, n)

for i ← 0 to n - 1 do

for j ← 0 to n - i - 1 do

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

// Swap arr[j] and arr[j + 1]

temp ← arr[j]

arr[j] ← arr[j + 1]

arr[j + 1] ← temp

end if

end for

end for

end procedure

**TASK 6 – Code for Bubble Sort**

class BubbleSort {  
  
 // Bubble Sort function  
 public static void bubbleSort(int[] arr) {  
 int n = arr.length;  
  
 // Outer loop for each pass  
 for (int i = 0; i < n - 1; i++) {  
 // Inner loop for comparison and swapping  
 for (int j = 0; j < n - i - 1; j++) {  
 // Swap if the current element is greater than the next  
 if (arr[j] > arr[j + 1]) {  
 // Swap arr[j] and arr[j+1]  
 int temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 }  
 }  
 }  
 }  
  
 // Utility function to print array  
 public static void printArray(int[] arr) {  
 for (int num : arr)  
 System.*out*.print(num + " ");  
 System.*out*.println();  
 }  
  
 // Main method to test  
 public static void main(String[] args) {  
 int[] arr = {64, 34, 25, 12, 22, 11, 90};  
 System.*out*.println("Original array:");  
 *printArray*(arr);  
  
 *bubbleSort*(arr);  
  
 System.*out*.println("Sorted array (Ascending):");  
 *printArray*(arr);  
 }  
}

**Output**

Original array:

64 34 25 12 22 11 90

Sorted array (Ascending):

11 12 22 25 34 64 90

**INSERTION SORT**

**TASK 7 – Algorithm for Insertion Sort**

1. Start from index 1 to n-1 (loop variable i):

Let initialize next = arr[i]

Initialize j = i - 1

2. While j >= 0 and arr[j] >next:

Move arr[j] one position ahead: arr[j + 1] = arr[j]

Decrement j by 1, ( j-- or j =j-1)

3. Place key at correct position: arr[j + 1] = next

4. Repeat steps 1 to 3 for all elements.

**Example**: i = 1 → next= 3

Compare with 5 → shift 5 → [5, 5, 4, 1]

Place next at j+1 → [3, 5, 4, 1]

i = 2 → next= 4

Compare with 5 → shift 5 → [3, 5, 5, 1]

Compare with 3 → no shift

Place next at j+1 → [3, 4, 5, 1]

**TASK 8 – PsuedoCode for Insertion Sort**

InsertionSort(arr, n)

for i from 1 to n - 1 do

next ← arr[i]

j ← i - 1

while j ≥ 0 and arr[j] > next

arr[j + 1] ← arr[j] // shift element to right

j ← j - 1

arr[j + 1] ← next

**TASK 9 – Code for Insertion Sort**

public class Day16\_InsertionSort {  
  
 public static void insertionSort(int[] arr) {  
 int n = arr.length;  
  
 for (int i = 1; i < n; i++) {  
 int next = arr[i]; // element to be positioned  
 int j = i - 1;  
  
 // Move elements of arr[0..i-1], that are greater than key, to one position ahead  
 while (j >= 0 && arr[j] > next) {  
 arr[j + 1] = arr[j];  
 j = j - 1;  
 }  
  
 arr[j + 1] = next;  
 }  
 }  
  
 public static void printArray(int[] arr) {  
 for (int value : arr) {  
 System.*out*.print(value + " ");  
 }  
 System.*out*.println();  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {64, 25, 12, 22, 11};  
  
 System.*out*.println("Original array:");  
 *printArray*(arr);  
  
 *insertionSort*(arr);  
  
 System.*out*.println("Sorted array (Ascending):");  
 *printArray*(arr);  
 }  
}

**Output**

Original array:

64 25 12 22 11

Sorted array (Ascending):

11 12 22 25 64

**TASK 10 – What are the Advantages and Disadvantages of Bubble sort?**

**Advantages of Bubble Sort**

* Simple to Understand and Implement
* No Additional Space Required
* Detects Sorted List Early

**Disadvantages of Bubble Sort**

* Time complexity is **O(n²)** in worst and average cases, making it very slow for large inputs.
* Performs a lot of **unnecessary swaps**, even when the list is only slightly unsorted.
* Much slower compared to advanced sorting algorithms like **Merge Sort**, **Quick Sort**, or **Heap Sort**.
* Its quadratic time complexity doesn’t scale well with increasing input sizes.

**TASK11 - 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);**

**}**

**Output**: public class RecLoop {

public int calc(int n) {

if (n == 0) return 0; // base case

return n + calc(n - 1); // reduce n to eventually reach 0

}

public static void main(String[] args) {

RecLoop r = new RecLoop();

int result = r.calc(5); // sum of 5 + 4 + 3 + 2 + 1 + 0

System.out.println("Result: " + result); // Output: 15

}

}

**MERGE SORT**

**TASK 12 – Algorithm for Merge Sort**

* If left < right, then:
* Find the middle point: mid = (left + right) / 2
* Recursively sort the first half: MergeSort(arr, left, mid)
* Recursively sort the second half: MergeSort(arr, mid + 1, right)
* Merge the two halves: Merge (arr, left, mid, right)
* Create an empty list `temp`
* Set: left = low, right = mid + 1
* Compares current values from both halves, puts the smaller one into temp.
* If right half is exhausted and left still has elements, add them.
* If left half is exhausted and right still has elements, add them.

**TASK13 – PseudoCode**

MergeSort(arr, left, right)

if left < right

mid ← (left + right) / 2

// Sort first half

MergeSort(arr, left, mid)

// Sort second half

MergeSort(arr, mid + 1, right)

// Merge the sorted halves

Merge(arr, left, mid, right)

Merge(arr, left, mid, right)

n1 ← mid - left + 1

n2 ← right - mid

Function merge(arr, low, mid, high):

Create an empty list temp

left ← low

right ← mid + 1

// Merge elements in sorted order

while left ≤ mid and right ≤ high:

if arr[left] ≤ arr[right]:

Add arr[left] to temp

left ← left + 1

else:

Add arr[right] to temp

right ← right + 1

// Copy any remaining elements from the left half

while left ≤ mid:

Add arr[left] to temp

left ← left + 1

// Copy any remaining elements from the right half

while right ≤ high:

Add arr[right] to temp

right ← right + 1

// Copy sorted elements from temp back to original array

for i from low to high:

arr[i] ← temp[i - low]

**TASK 14 – Code for MergeSort**

import java.util.\*;  
  
class mergeSort {  
 private static void merge(int[] arr, int low, int mid, int high) {  
 ArrayList<Integer> temp = new ArrayList<>(); // temporary array  
 int left = low; // starting index of left half of arr  
 int right = mid + 1; // starting index of right half of arr  
  
 //storing elements in the temporary array in a sorted manner//  
  
 while (left <= mid && right <= high) {  
 if (arr[left] <= arr[right]) {  
 temp.add(arr[left]);  
 left++;  
 } else {  
 temp.add(arr[right]);  
 right++;  
 }  
 }  
  
 // if elements on the left half are still left //  
  
 while (left <= mid) {  
 temp.add(arr[left]);  
 left++;  
 }  
  
 // if elements on the right half are still left //  
 while (right <= high) {  
 temp.add(arr[right]);  
 right++;  
 }  
  
 // transfering all elements from temporary to arr //  
 for (int i = low; i <= high; i++) {  
 arr[i] = temp.get(i - low);  
 }  
 }  
  
 public static void MergeSort(int[] arr, int low, int high) {  
 if (low >= high) return;  
 int mid = (low + high) / 2 ;  
 *MergeSort*(arr, low, mid); // left half  
 *MergeSort*(arr, mid + 1, high); // right half  
 *merge*(arr, low, mid, high); // merging sorted halves  
 }  
}  
class tUf {  
 public static void main(String args[]) {  
 Scanner sc = new Scanner(System.*in*);  
 int n = 7;  
 int arr[] = { 9, 4, 7, 6, 3, 1, 5 };  
 System.*out*.println("Before sorting array: ");  
 for (int i = 0; i < n; i++) {  
 System.*out*.print(arr[i] + " ");  
 }  
 System.*out*.println();  
 mergeSort.*MergeSort*(arr, 0, n - 1);  
 System.*out*.println("After sorting array: ");  
 for (int i = 0; i < n; i++) {  
 System.*out*.print(arr[i] + " ");  
 }  
 System.*out*.println();  
 }  
  
}

**Output**

Before sorting array:

9 4 7 6 3 1 5

After sorting array:

1 3 4 5 6 7 9

**TASK 15 – Algorithm for Quick Sort**

If low < high:

* Find the partition index : pivotIndex = Partition (arr, low, high)
* Recursively sort the left subarray: QuickSort(arr, low, pivotIndex - 1)
* Recursively sort the right subarray: QuickSort(arr, pivotIndex + 1, high)

1. Set pivot = arr [high] // Last element as pivot

2. Initialize i = low - 1

3. For j = low to high - 1:

a. If arr[j] < pivot:

i = i + 1

Swap arr[i] and arr[j]

4. After loop, swap arr[i + 1] and arr[high] (pivot)

5. Return i + 1 as the partition index

**TASK16 – PseudoCode for QuickSort**

Function QuickSort(arr, low, high):

if low < high:

pivotIndex ← Partition(arr, low, high)

// Recursively sort elements before pivot

QuickSort(arr, low, pivotIndex - 1)

// Recursively sort elements after pivot

QuickSort(arr, pivotIndex + 1, high)

Function Partition(arr, low, high):

pivot ← arr[high] // Choose last element as pivot

i ← low - 1 // Index of smaller element

for j ← low to high - 1:

if arr[j] < pivot:

i ← i + 1

Swap arr[i] and arr[j]

Swap arr[i + 1] and arr[high] // Place pivot in correct position

return i + 1 // Return partition index

**TASK17 – Code for QuickSort**

public class Day16\_QuickSort {  
  
 public void sort(int[] arr, int low, int high) {  
 if (low < high) {  
 int pi = partition(arr, low, high);  
  
 // Recursively sort elements before partition and after partition  
 sort(arr, low, pi - 1);  
 sort(arr, pi + 1, high);  
 }  
 }  
  
 // This method takes last element as pivot, places the pivot element at its correct position in sorted array,  
 // and places all smaller (than pivot) to left of pivot and all greater elements to right of pivot  
 private int partition(int[] arr, int low, int high) {  
 int pivot = arr[high]; // Choosing the last element as pivot  
 int i = (low - 1); // Index of the smaller element; keeps track of where to place the next smaller-than-pivot number.  
  
  
 for (int j = low; j < high; j++) {  
 // If current element is smaller than or equal to pivot  
 if (arr[j] <= pivot) {  
 i++;  
  
 // Swap arr[i] and arr[j]  
 int temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
 }  
  
 // Swap arr[i+1] and arr[high] (or pivot)  
 int temp = arr[i + 1];  
 arr[i + 1] = arr[high];  
 arr[high] = temp;  
  
 return i + 1;  
 }  
  
 // Utility method to print the array  
 public static void printArray(int[] arr) {  
 for (int i = 0; i < arr.length; ++i) {  
 System.*out*.print(arr[i] + " ");  
 }  
 System.*out*.println();  
 }  
  
 // Driver code to test the QuickSort implementation  
 public static void main(String[] args) {  
 int[] arr = {10, 7, 8, 9, 1, 5};  
 int n = arr.length;  
  
 Day16\_QuickSort qs = new Day16\_QuickSort();  
 System.*out*.println("Original array:");  
 *printArray*(arr);  
  
 qs.sort(arr, 0, n - 1);  
  
 System.*out*.println("Sorted array:");  
 *printArray*(arr);  
 }  
}

**Output**

Original array:

10 7 8 9 1 5

Sorted array:

1 5 7 8 9 10