

**Bubble Sort**

**ALGORITHM** BubbleSort( $A[0..n-1]$ )  
 // Sorts a given array using bubble sort  
 // Input: An array  $A[0..n-1]$  of orderable  
 // elements  
 // Output: Array  $A[0..n-1]$  sorted in  
 // ascending order  
**for**  $i \leftarrow 0$  **to**  $n-2$  **do**  
   **for**  $j \leftarrow 0$  **to**  $n-2-i$  **do**  
**if**  $A[j+1] < A[j]$   
   swap  $A[j]$  and  $A[j+1]$

**Selection Sort**

**ALGORITHM** SelectionSort( $A[0..n-1]$ )  
 // Sorts a given array using selection sort  
 // Input: An array  $A[0..n-1]$  of orderable  
 // elements  
 // Output: Array  $A[0..n-1]$  sorted in  
 // ascending order  
**for**  $i \leftarrow 0$  **to**  $n-2$  **do**  
    $\text{min} \leftarrow i$   
   **for**  $j \leftarrow i+1$  **to**  $n-1$  **do**  
**if**  $A[j] < A[\text{min}]$   
    $\text{min} \leftarrow j$   
   swap  $A[i]$  and  $A[\text{min}]$

**Insertion Sort**

**ALGORITHM** InsertionSort( $A[0..n-1]$ )  
 // Sorts a given array using insertion sort  
 // Input: An array  $A[0..n-1]$  of orderable  
 // elements  
 // Output: Array  $A[0..n-1]$  sorted in  
 // ascending order  
**for**  $i \leftarrow 1$  **to**  $n-1$  **do**  
    $v \leftarrow A[i]$   
    $j \leftarrow i-1$   
   **while**  $j \geq 0$  **and**  $A[j] > v$  **do**  
    $A[j+1] \leftarrow A[j]$   
    $j \leftarrow j-1$   
    $A[j+1] \leftarrow v$

**Merge Sort**

**ALGORITHM** MergeSort( $A[0..n-1]$ )  
 // Sorts a given  $A[0..n-1]$  by recursive mergesort  
 // Input: An array  $A[0..n-1]$  of orderable elements  
 // Output: Array  $A[0..n-1]$  sorted in nondecreasing order  
**if**  $n > 1$   
   copy  $A[0.. \lfloor n/2 \rfloor - 1]$  to  $B[0.. \lfloor n/2 \rfloor - 1]$   
   copy  $A[\lfloor n/2 \rfloor .. n-1]$  to  $C[0.. \lfloor n/2 \rfloor - 1]$   
   MergeSort( $B[0.. \lfloor n/2 \rfloor - 1]$ )  
   MergeSort( $C[0.. \lfloor n/2 \rfloor - 1]$ )  
   Merge( $B, C, A$ )

**ALGORITHM** Merge( $B[0..p-1], C[0..q-1], A[0..p+q-1]$ )  
 // Merges two sorted arrays into one sorted array  
 // Input: Arrays  $B[0..p-1]$  and  $C[0..q-1]$  both sorted  
 // Output: Sorted array  $A[0..p+q-1]$  of the elements of  $B$  and  $C$   
 $i \leftarrow 0$   
 $j \leftarrow 0$   
 $k \leftarrow 0$   
**while**  $i < p$  **and**  $j < q$  **do**  
   **if**  $B[i] \leq C[j]$   
    $A[k] \leftarrow B[i]$   
    $i \leftarrow i+1$   
   **else**  
    $A[k] \leftarrow C[j]$   
    $j \leftarrow j+1$   
    $k \leftarrow k+1$   
**if**  $i = p$   
   copy  $C[j..q-1]$  to  $A[k..p+q-1]$   
**else**  
   copy  $B[i..p-1]$  to  $A[k..p+q-1]$

**Quick Sort**

**ALGORITHM** QuickSort( $A[l..r]$ )  
 // Sorts a subarray by quicksort  
 // Input: A subarray  $A[l..r]$  of  $A[0..n-1]$ , defined by its left and right indices  $l$  and  $r$   
 // Output: Subarray  $A[l..r]$  sorted in nondecreasing order  
**if**  $l < r$   
    $s \leftarrow \text{Partition}(A[l..r])$   
   QuickSort( $A[l..s-1]$ )  
   QuickSort( $A[s+1..r]$ )

**ALGORITHM** Partition( $A[l \dots r]$ )

// Partitions a subarray by using its first element as a pivot

// Input: A subarray  $A[l \dots r]$  of  $A[0 \dots n-1]$ , defined by its left and right indices  $l$  and  $r$  ( $l < r$ )

// Output: Subarray  $A[l \dots r]$ , with split position returned as this functions value

$p \leftarrow A[l]$

$i \leftarrow l$

$j \leftarrow r + 1$

**repeat**

**repeat**  $i \leftarrow i + 1$  **until**  $A[i] \geq p$

**repeat**  $j \leftarrow j - 1$  **until**  $A[j] \leq p$

    swap( $A[i]$  and  $A[j]$ )

**until**  $i \geq j$

swap ( $A[i]$ ,  $A[j]$ )

swap ( $A[l]$ ,  $A[j]$ )

**return**  $j$

**Heap Sort**

**Step 01:** Heap construction – construct a heap for a given array

**Step 02:** Maximum Deletions – Apply the root-deletion operation  $n-1$  times to the remaining heap

**Algorithm for Heap Construction****ALGORITHM** HeapBottomUp( $H[1..n]$ )

// Constructs a heap from the elements of a given array by the bottom up algorithm

// Input: An array  $H[1..n]$  of orderable items

// Output: A heap  $H[1..n]$

**for**  $i \leftarrow \lfloor n/2 \rfloor$  **downto** 1 **do**

$k \leftarrow i$

$v \leftarrow H[k]$

    heap  $\leftarrow$  false

**while not** heap **and**  $2 * k \leq n$  **do**

$j \leftarrow 2 * k$

**if**  $j < n$

**if**  $H[j] < H[j + 1]$

$j \leftarrow j + 1$

**if**  $v \geq H[j]$

            heap  $\leftarrow$  true

**else**

$H[k] \leftarrow H[j]$

$K \leftarrow j$

$H[k] \leftarrow v$

**Algorithm for maximum key deletion**

**Step 01:** Exchange the root's key with the last key  $K$  of the Heap

**Step 02:** Decrease the heap's size by 1

**Step 03:** "Heapify" the smaller tree