Data Structures and Algorithms

SORTING ALGORITHMS HANDOUT

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 ← 0 **to** n - 2 **do**
  for j ← 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 ← 0 **to** n - 2 **do**
  min ← i
  for j ← i + 1 **to** n - 1 **do**
    if A[j] < A[min]
      min ← j
  swap A[i] and A[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 ← 1 **to** n - 1 **do**
  v ← A[i]
  j ← i - 1
  while j >= 0 **and** A[j] > v **do**
    A[j + 1] ← A[j]
    j ← j - 1
  A[j + 1] ← 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… |_n/2_| - 1] to B[0… |_n/2_| - 1]
  copy A[|_n/2_| … n - 1] to C[0…… |¯ n/2 ¯| - 1]
  MergeSort(B[0… |_n/2_| - 1])
  MergeSort(C[0…… |¯ n/2 ¯| - 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 ← 0
j ← 0
k ← 0
while i < p **and** j < q **do**
  if B[i] <= C[j]
    A[k] ← B[i]
    i ← i + 1
  else
    A[k] ← C[j]
    j ← j + 1
  k ← 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 ← Partition(A[l…r])
  QuickSort(A[l… s - 1])
  QuickSort(A[s + 1…r])

```
ALGORITHM Partition(A[l… r])
// Partitions a subarray by using its first element as a pivot
// Input: A subarray A[l… r] of A[0… n-1], defined by its left and right indices l and r (l < r)
// Output: Subarray A[l… r], with split position returned as this functions value
p ← A[l]
i ← l
j ← r + 1
repeat
   repeat i ← i + 1 until A[i] >= p
   repeat j ← j – 1 until A[j] <= p
   swap(A[i] and A[j])
until i >= 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 ← |_ n/2 _| downto 1 do
    k ← i
    v ← H[k]
    heap ← false
    while not heap and 2 * k <=n do
        j ← 2 * k
        if j < n
            if H[j] < H[j + 1]
                j ← j + 1
        if v >= H[ j ]
            heap ← true
        else
            H[k] ← H[j]
            K ← j
    H[k] ← 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