## Insertion Sort

### Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 – If it is the first element, it is already sorted. return 1;

Step 2 – Pick next element

Step 3 – Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the

value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until list is sorted

### Pseudocode

```
procedure insertionSort( A : array of items )
  int holePosition
  int valueToInsert
  for i = 1 to length(A) inclusive do:
    /* select value to be inserted */
    valueToInsert = A[i]
    holePosition = i
      /*locate hole position for the element to be inserted */
    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition -1
```

```
end while

/* insert the number at hole position */
A[holePosition] = valueToInsert
  end for
end procedure
```

**Selection Sort**

**Algorithm**

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

Step 4 – Increment MIN to point to next element

Step 5 – Repeat until list is sorted

**Pseudocode**

```
procedure selection sort
  list  : array of items
  n     : size of list
 for i = 1 to n - 1
/* set current element as minimum*/
  min = i
   /* check the element to be minimum */
 for j = i+1 to n
   if list[j] < list[min] then
```

```
        min = j;
      end if
    end for
    /* swap the minimum element with the current element*/
    if indexMin != i  then
      swap list[min] and list[i]
    end if
  end for
end procedure
```

**Bubble Sort**

**Algorithm**

```
begin BubbleSort(list)
  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for
  return list
end BubbleSort
```

**Pseudocode**

```
procedure bubbleSort( list : array of items )
 loop = list.count;
  for i = 0 to loop-1 do:
  swapped = false
        for j = 0 to loop-1 do:
     /* compare the adjacent elements */
   if list[j] > list[j+1] then
    /* swap them */
    swap( list[j], list[j+1] )
    swapped = true
  end if
     end for
   /*if no number was swapped that means
array is sorted now, break the loop.*/
   if(not swapped) then
 break
end if
 end for
end procedure return list
```

## Merge Sort

### Algorithm

Step 1 – if it is only one element in the list it is already sorted, return.

Step 2 – divide the list recursively into two halves until it can no more be divided.

Step 3 – merge the smaller lists into new list in sorted order.

### Pseudocode

```
procedure mergesort( var a as array )
  if ( n == 1 ) return a
  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]
  l1 = mergesort( l1 )
  l2 = mergesort( l2 )
  return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )
  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
```

```
end if
end while
while ( a has elements )
add a[0] to the end of c
remove a[0] from a
end while
while ( b has elements )
add b[0] to the end of c
remove b[0] from b
end while
return c
end procedure
```

**Quick Sort**

**Algorithm**

***Quick Sort Pivot selection Algorithm ()***

Step 1 – Choose the highest index value has pivot

Step 2 – Take two variables to point left and right of the list excluding pivot

Step 3 – left points to the low index

Step 4 – right points to the high

Step 5 – while value at left is less than pivot move right

Step 6 – while value at right is greater than pivot move left

Step 7 – if both step 5 and step 6 does not match swap left and right

Step 8 – if left ≥ right, the point where they met is new pivot

***Quick Sort Algorithm ()***

Step 1 – Make the right-most index value pivot

Step 2 – partition the array using pivot value

Step 3 – quicksort left partition recursively

Step 4 – quicksort right partition recursively

**Pseudocode ()**

***Quick Sort Pivot selection ()***

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1
   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while
      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while
      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if
   end while
```

```
   swap leftPointer,right
   return leftPointer
end function
```

***Quick Sort Pivot selection ()***

```
procedure quickSort(left, right)
  if right-left <= 0
    return
  else
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
  end if
end procedure
```

**Heap Sort**

**Algorithm**

***HEAPSORT(A)***

BUILD-HEAP(A)

**for** i ← length[A] **downto** 2

  **do** exchange A[1] ↔ A[i]

    heap-size[A] ← heap-size[A] -1

   HEAPIFY(A, 1)

***BUILD-HEAP(A)***

heap-size[A] ← length[A]

**for** i ← ⌊length[A]/2⌋ **downto** 1

**do** HEAPIFY(A, i)

**HEAPIFY(*A*, *i*)**

*l* ← LEFT(*i*)

*r* ← RIGHT(*i*)

**if** *l* ≤ *heap-size*[*A*] and *A*[*l*] > *A*[*i*]

  **then** largest ← l

  **else** *largest* ← *i*

**if** *r* ≤ *heap-size*[*A*] and *A*[*r*] > *A*[*largest*]

  **then** *largest* ← *r*

**if** *largest* ≠ *i*

**then** exchange $A[i] \leftrightarrow A[largest]$

HEAPIFY($A$,*largest*)

**Pseudo code**

```
Heapsort(A) {
  BuildHeap(A)
  for i <- length(A) downto 2 {
    exchange A[1] <-> A[i]
    heapsize <- heapsize -1
    Heapify(A, 1)
}

BuildHeap(A) {
  heapsize <- length(A)
  for i <- floor( length/2 ) downto 1
    Heapify(A, i)
}
Heapify(A, i) {
  le <- left(i)
  ri <- right(i)
  if (le<=heapsize) and (A[le]>A[i])
    largest <- le
  else
    largest <- i
```

```
if (ri<=heapsize) and (A[ri]>A[largest])
  largest <- ri
if (largest != i) {
  exchange A[i] <-> A[largest]
  Heapify(A, largest)
 }
}
```

**Linear Search**

**Algorithm**

Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

**Pseudo code**

```
procedure linear_search (list, value)
 for each item in the list
  if match item == value
   return the item's location
 end if
end for
end procedure
```

**Binary Search**

**Algorithm**

Step 1 – Start searching data from middle of the list.

Step 2 – If it is a match, return the index of the item, and exit.

Step 3 – If it is not a match, probe position.

Step 4 – Divide the list using probing formula and find the new midle.

Step 5 – If data is greater than middle, search in higher sub-list.

Step 6 – If data is smaller than middle, search in lower sub-list.

Step 7 – Repeat until match.

**Pseudo code**

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched
  Set lowerBound = 1
  Set upperBound = n
  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.
      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
        if A[midPoint] < x
      set lowerBound = midPoint + 1
        if A[midPoint] > x
      set upperBound = midPoint - 1
```

```
  if A[midPoint] = x
    EXIT: x found at location midPoint
end while
end procedure
```