## 

## How Insertion Sort Works?

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

These values are not in a sorted order.

Insertion Sort

So we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

Insertion Sort

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of 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.

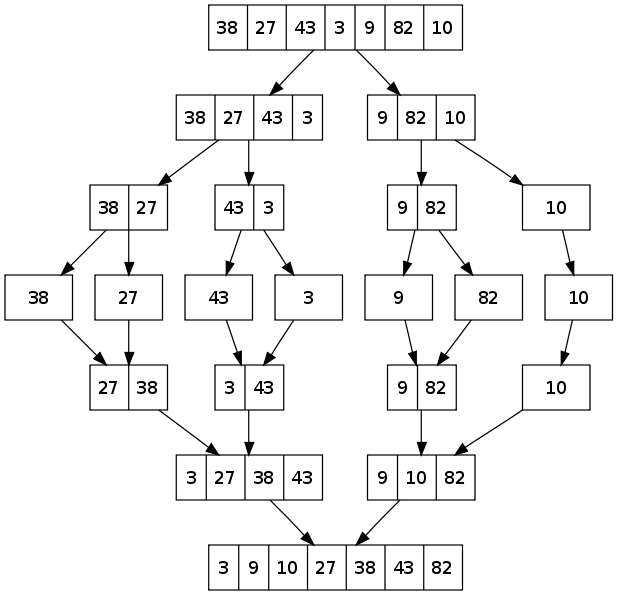
**Step 1** − If it is the first element, it is already sorted. return 1;  
**Step 2** − Pick next element  
**Step 3** − Compare with all elements in the sorted sub-list  
**Step 4** − Shift all the elements in the sorted sub-list that is greater than the   
 value to be sorted  
**Step 5** − Insert 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

**MergeSort(arr[], l, r)**  
If r > l  
 **1.** Find the middle point to divide the array into two halves:   
 middle m = (l+r)/2  
 **2.** Call mergeSort for first half:   
 Call mergeSort(arr, l, m)  
 **3.** Call mergeSort for second half:  
 Call mergeSort(arr, m+1, r)  
 **4.** Merge the two halves sorted in step 2 and 3:  
 Call merge(arr, l, m, r)

The following diagram from [wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg) shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



## How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

Bubble Sort

Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

Bubble Sort

We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

Bubble Sort

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

## Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

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

## How Selection Sort Works?

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of 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