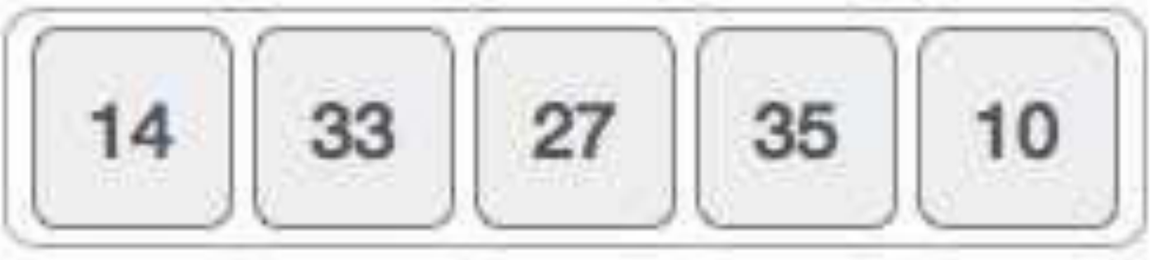
Bubble Sorting:

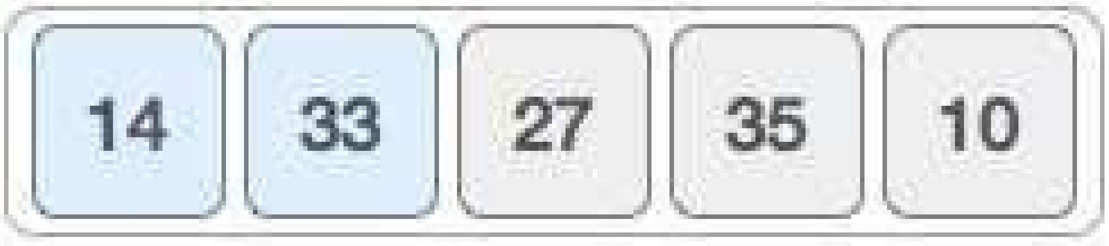
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Bubble Sort Working Concept:

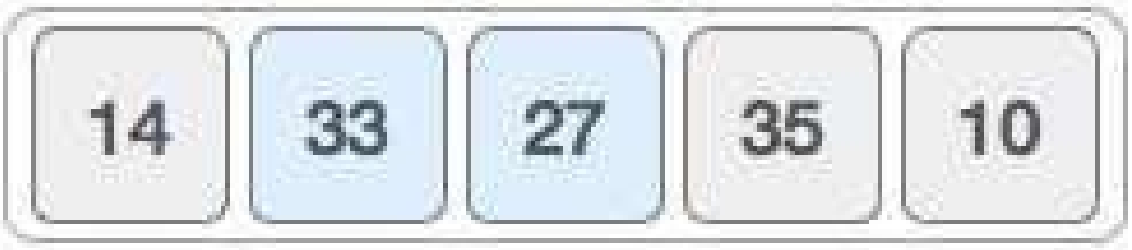
We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



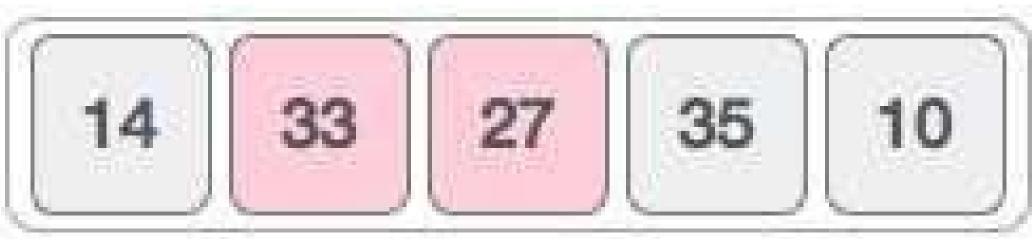
Bubble sort starts with very first two elements, comparing them to check which one is greater.



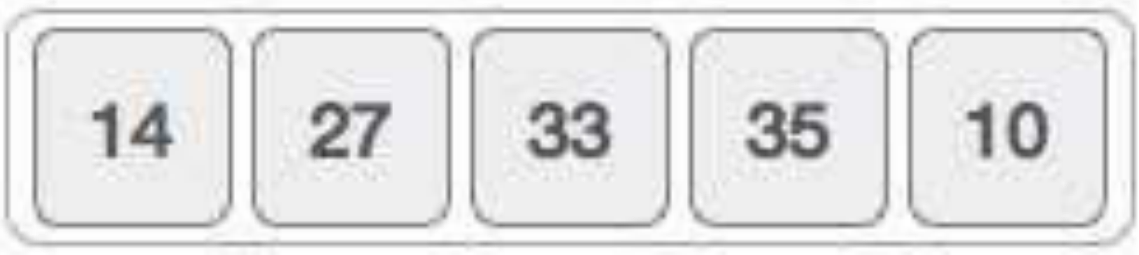
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



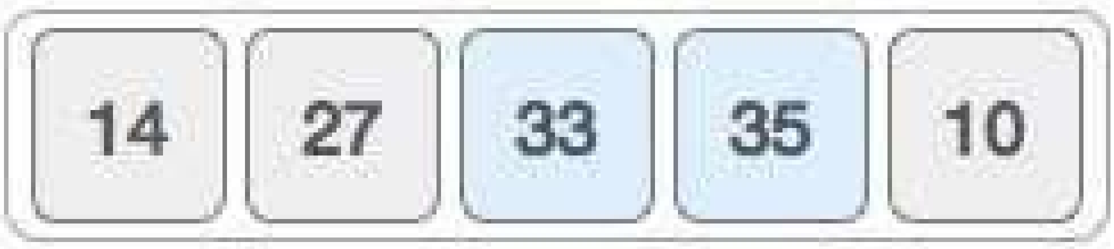
We find that 27 is smaller than 33 and these two values must be swapped.



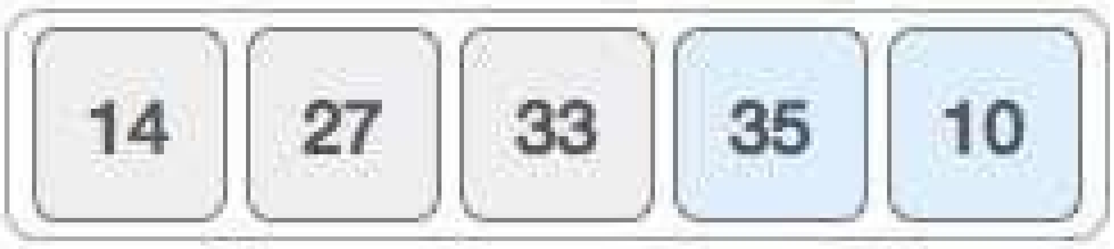
The new array should look like this −



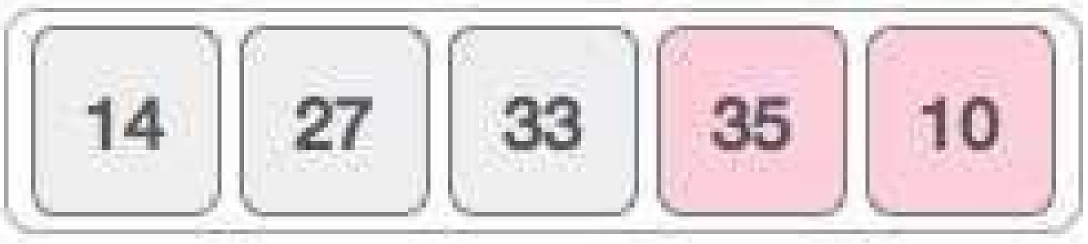
Next we compare 33 and 35. We find that both are in already sorted positions.



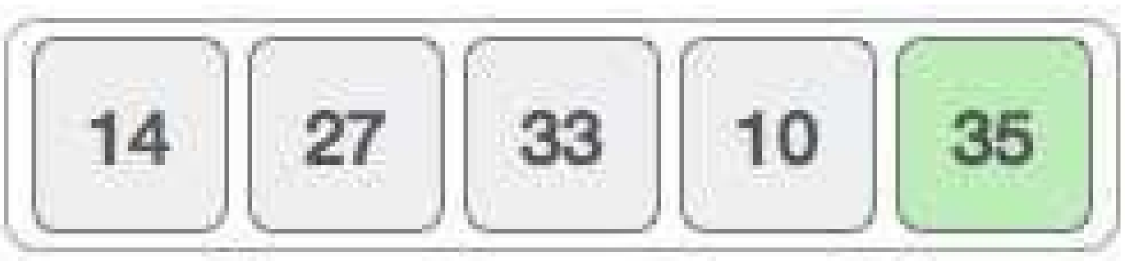
Then we move to the next two values, 35 and 10.



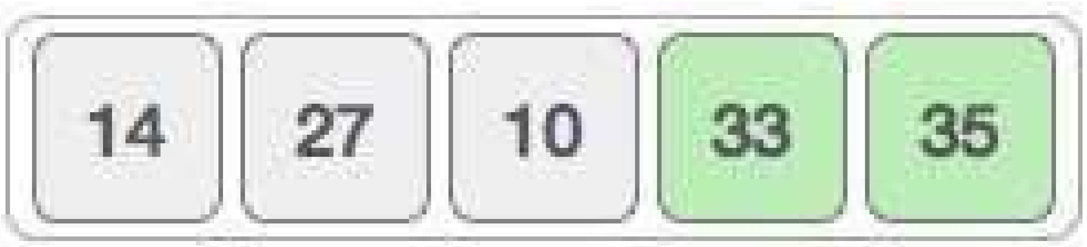
We know then that 10 is smaller 35. Hence they are not sorted.



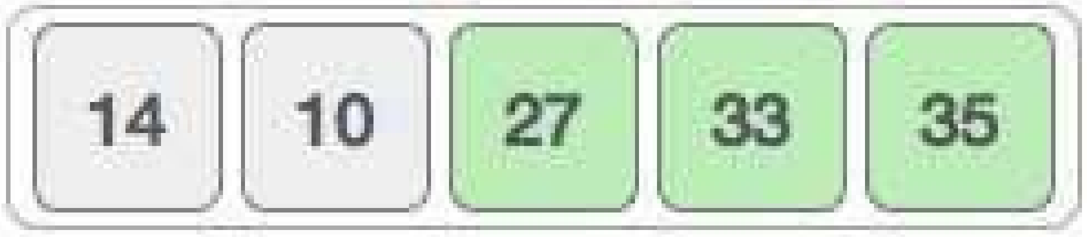
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



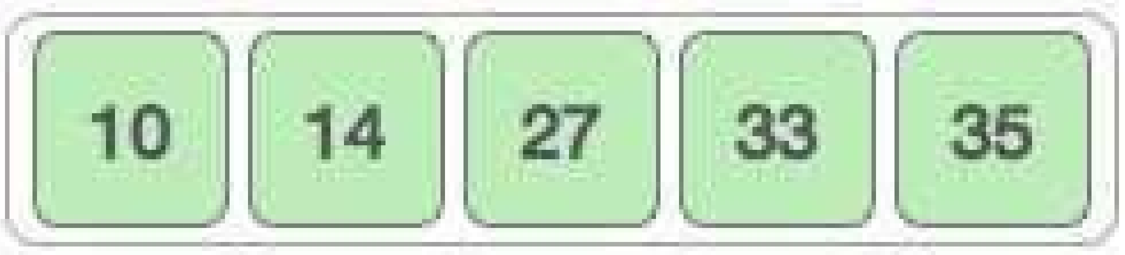
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



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



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



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
```

Bubble Sort Program in C:

Implementation in C

```
#include <stdio.h>
#include <stdbool.h>


#define MAX 10
int list[MAX] = {1,8,4,6,0,3,5,2,7,9};


void display(){
   int i;
   printf("[");
   // navigate through all items
   for(i = 0; i < MAX; i++){
      printf("%d ",list[i]);
   }

   printf("]\n");
}
void bubbleSort() {
   int temp;
   int i,j;

   bool swapped = false;

   // loop through all numbers
   for(i = 0; i < MAX-1; i++) {
      swapped = false;

      // loop through numbers falling
      ahead for(j = 0; j < MAX-1-i; j++) {
         printf("     Items compared: [ %d, %d ] ", list[j],list[j+1]);

         // check if next number is lesser than current no
         //   swap the numbers.
         //  (Bubble up the highest number)

         if(list[j] > list[j+1]) {
            temp = list[j];
            list[j] = list[j+1];
            list[j+1] = temp;

            swapped = true;
            printf(" => swapped [%d, %d]\n",list[j],list[j+1]);
         }else {
            printf(" => not swapped\n");
         }
```

```
        }

        // if no number was swapped that means
        //   array  is  sorted  now,  break  the
        loop. if(!swapped) {
           break;
        }

        printf("Iteration %d#: ",(i+1));
        display();
     }

  }

  main(){
     printf("Input Array: ");
     display(); printf("\n");

     bubbleSort();
     printf("\nOutput Array:
     "); display();
  }
```

If we compile and run the above program, it will produce the following result –

```
Input Array: [1 8 4 6 0 3 5 2 7 9 ]
     Items compared: [ 1, 8 ]  => not swapped
     Items compared: [ 8, 4 ]  => swapped [4, 8]
     Items compared: [ 8, 6 ]  => swapped [6, 8]
     Items compared: [ 8, 0 ]  => swapped [0, 8]
     Items compared: [ 8, 3 ]  => swapped [3, 8]
     Items compared: [ 8, 5  ] => swapped [5, 8]
     Items compared: [ 8, 2  ] => swapped [2, 8]
     Items compared: [ 8, 7  ] => swapped [7, 8]
     Items compared: [ 8, 9  ] => not swapped
```

```
Iteration 1#: [1 4 6  0  3 5 2 7 8 9 ]
     Items compared: [ 1, 4 ]  => not swapped
     Items compared: [ 4, 6 ]  => not swapped
     Items compared: [ 6, 0 ]  => swapped [0, 6]
     Items compared: [ 6, 3 ]  => swapped [3, 6]
     Items compared: [ 6, 5 ]  => swapped [5, 6]
     Items compared: [ 6, 2 ]  => swapped [2, 6]
     Items compared: [ 6, 7 ]  => not swapped
     Items compared: [ 7, 8 ]  => not swapped
Iteration 2#: [1 4 0 3 5 2 6 7 8 9 ]
     Items compared: [ 1, 4 ]  => not swapped
     Items compared: [ 4, 0 ]  => swapped [0, 4]
     Items compared: [ 4, 3 ]  => swapped [3, 4]
     Items compared: [ 4, 5 ]  => not swapped
     Items compared: [ 5, 2 ]  => swapped [2, 5]
     Items compared: [ 5, 6 ]  => not swapped
     Items compared: [ 6, 7 ]  => not swapped
Iteration 3#: [1 0 3 4 2 5 6 7 8 9 ]
     Items compared: [ 1, 0 ]  => swapped [0, 1]
     Items compared: [ 1, 3 ]  => not swapped
     Items compared: [ 3, 4 ]  => not swapped
     Items compared: [ 4, 2 ]  => swapped [2, 4]
     Items compared: [ 4, 5 ]  => not swapped
     Items compared: [ 5, 6 ]  => not swapped
Iteration 4#: [0 1 3 2 4 5 6 7 8 9 ]
     Items compared: [ 0, 1 ]  => not swapped
     Items compared: [ 1, 3 ]  => not swapped
     Items compared: [ 3, 2 ]  => swapped [2, 3]
     Items compared: [ 3, 4 ]  => not swapped
     Items compared: [ 4, 5 ]  => not swapped
Iteration 5#: [0 1 2 3 4 5 6 7 8 9 ]
     Items compared: [ 0, 1 ]  => not swapped
     Items compared: [ 1, 2 ]  => not swapped
     Items compared: [ 2, 3 ]  => not swapped
     Items compared: [ 3, 4 ]  => not swapped
Output Array: [0 1 2 3 4 5 6 7 8 9 ]
```

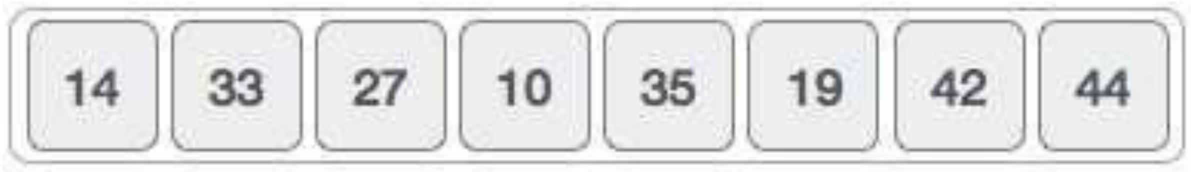
Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

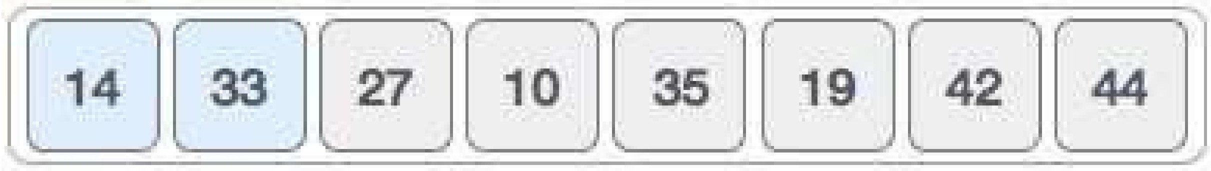
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2), where **n** is the number of items.

Insertion Sort Working:

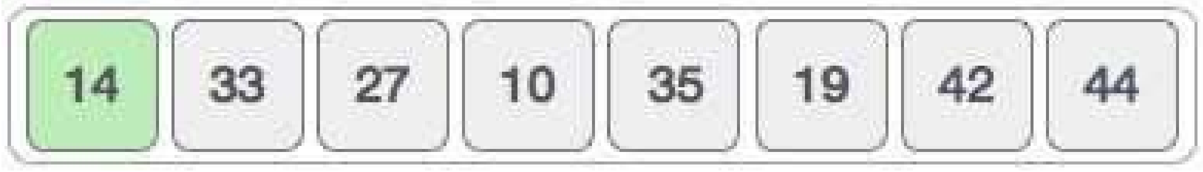
We take an unsorted array for our example.



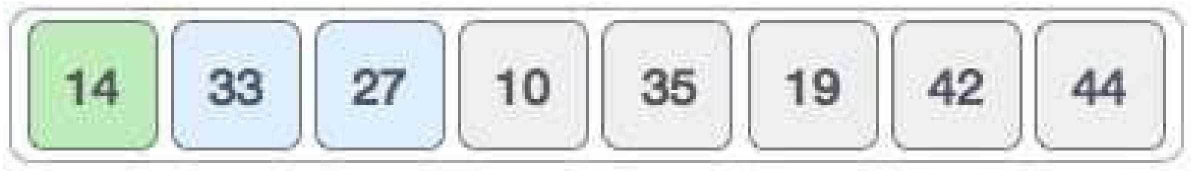
Insertion sort compares the first two elements.



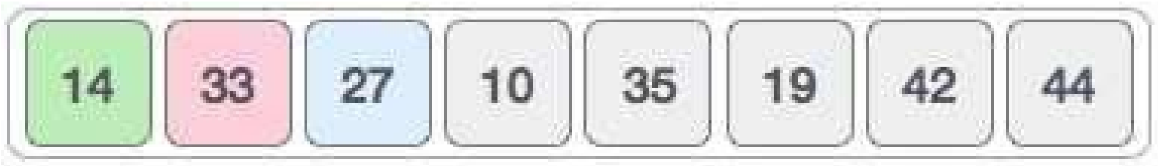
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



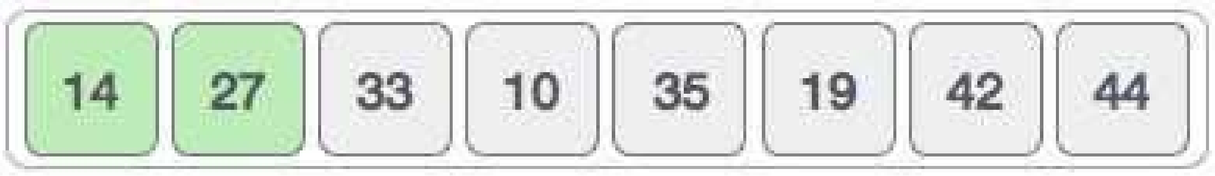
Insertion sort moves ahead and compares 33 with 27.



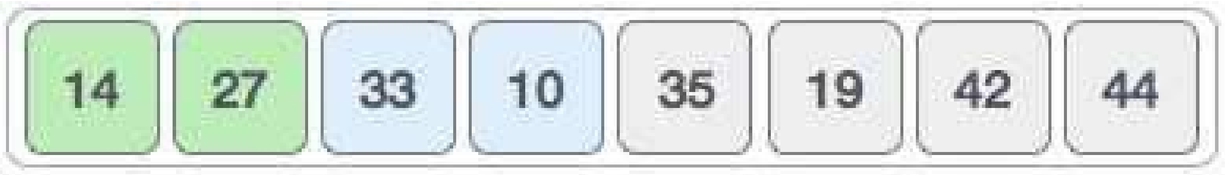
And finds that 33 is not in the correct position.



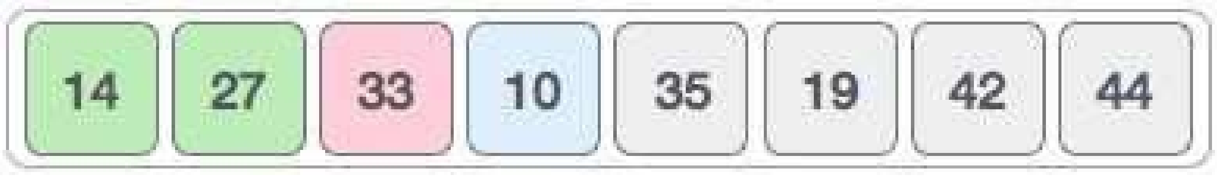
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



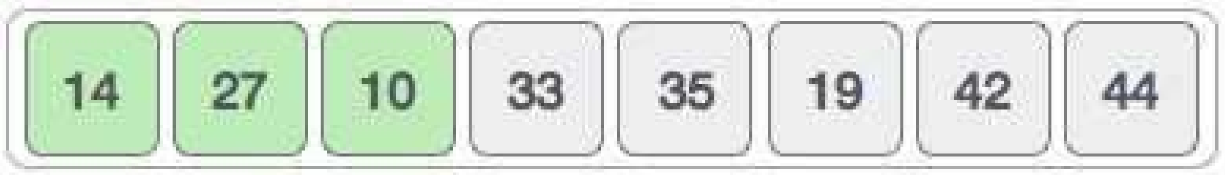
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



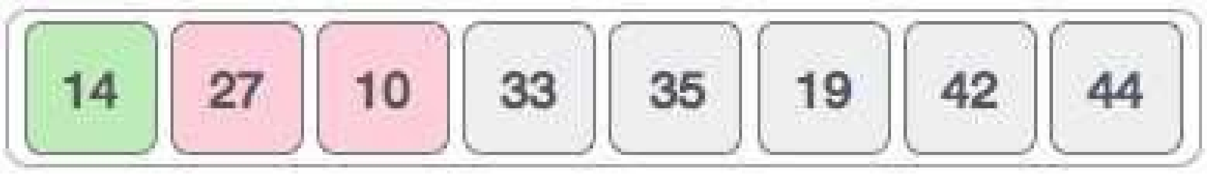
These values are not in a sorted order.



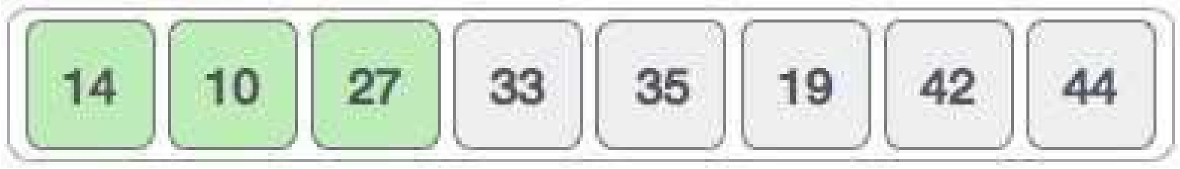
So we swap them.



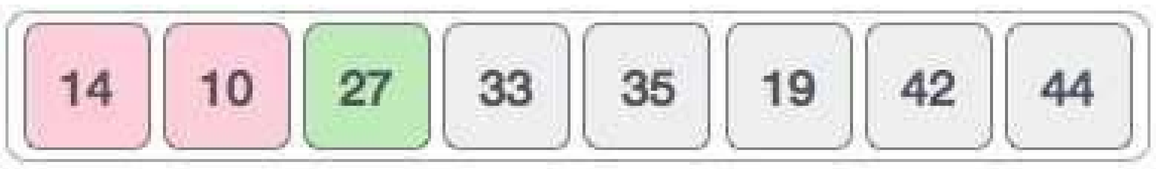
However, swapping makes 27 and 10 unsorted.



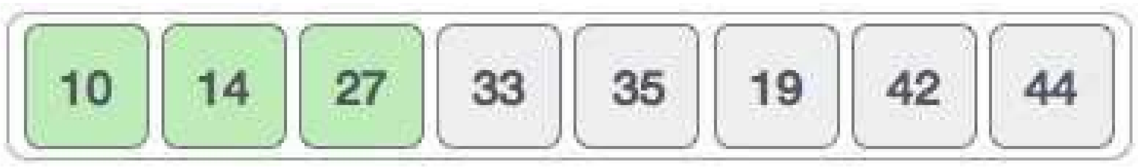
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



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



This process goes on until all the unsorted values are covered in a sorted sub-list.

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
```

Insertion Sort Program in C:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int
   count){ int i;

   for(i = 0;i <count-1;i++){
      printf("=");
   }
   printf("=\n");
}

void display(){
   int i;
   printf("[");

   // navigate through all
   items for(i = 0;i<MAX;i++){
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void insertionSort(){
   int valueToInsert;
   int holePosition;
   int i;

   // loop through all numbers
   for(i = 1; i < MAX; i++){

      // select a value to be inserted.
      valueToInsert = intArray[i];

      // select the hole position where number is to be inserted
      holePosition = i;

      // check if previous no. is larger than value to be inserted
      while (holePosition > 0 && intArray[holePosition-1] > valueToInsert){
         intArray[holePosition] = intArray[holePosition-1];
         holePosition--;
         printf(" item moved : %d\n" , intArray[holePosition]);
      }
      if(holePosition != i){
         printf(" item inserted : %d, at position : %d\n" ,
valueToInsert,holePosition);
         // insert the number at hole position
         intArray[holePosition] = valueToInsert;
      }

      printf("Iteration %d#:",i);
      display();

   }
}

main(){
   printf("Input Array: ");
   display();
   printline(50);
   insertionSort();
   printf("Output Array:
   "); display();
   printline(50);
}
```

If we compile and run the above program, it will produce the following result –

```
Input Array: [4, 6, 3, 2, 1, 9, 7]
==================================================
iteration 1#: [4, 6, 3, 2, 1, 9, 7]
  item moved :6
  item moved :4
  item inserted :3, at position :0
iteration 2#: [3, 4, 6, 2, 1, 9, 7]
  item moved :6
  item moved :4
  item moved :3
  item inserted :2, at position :0
iteration 3#: [2, 3, 4, 6, 1, 9, 7]
  item moved :6
  item moved :4
  item moved :3
  item moved :2
  item inserted :1, at position :0
iteration 4#: [1, 2, 3, 4, 6, 9, 7]
  iteration 5#: [1, 2, 3, 4, 6, 9,
  7] item moved :9
  item moved :6
  item inserted :7, at position :4
iteration 6#: [1, 2, 3, 4, 7, 6, 9]
Output Array: [1, 2, 3, 4, 7, 6, 9]
==================================================
```

Selection sort:

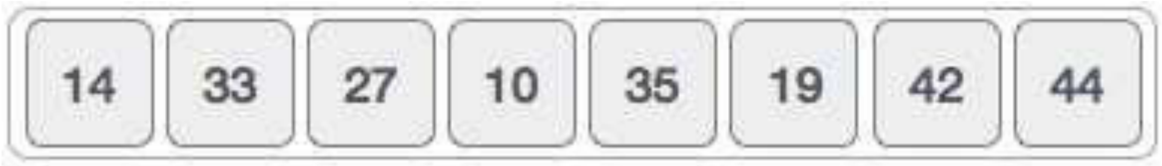
Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

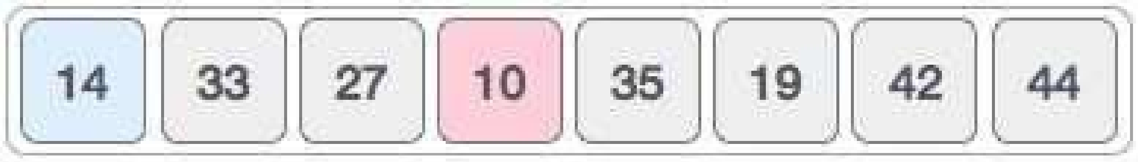
This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

Selection Sort Working:

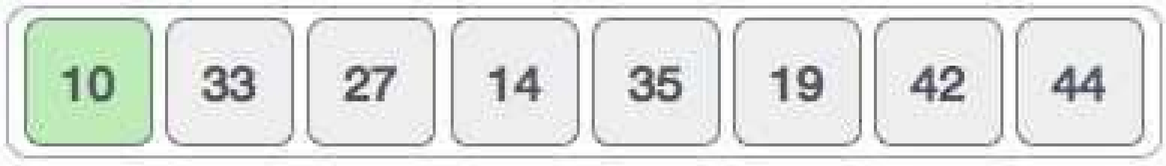
Consider the following depicted array as an example.



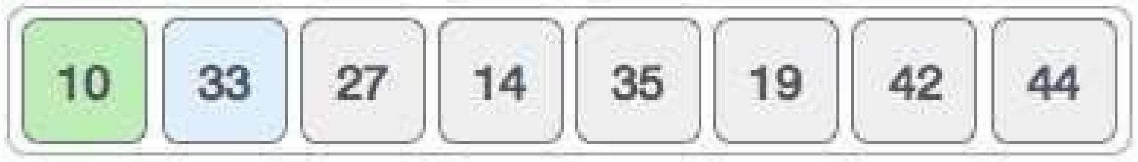
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



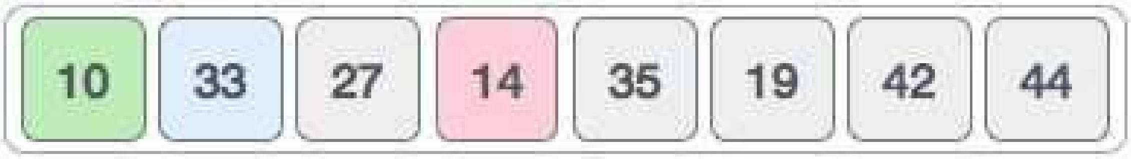
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



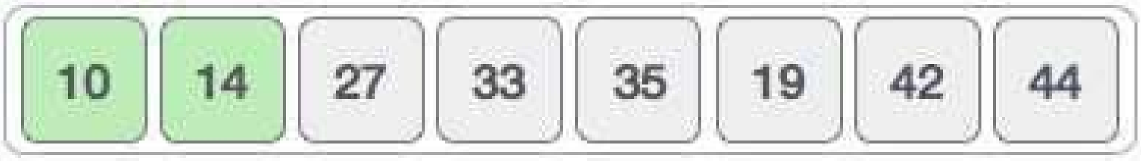
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



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

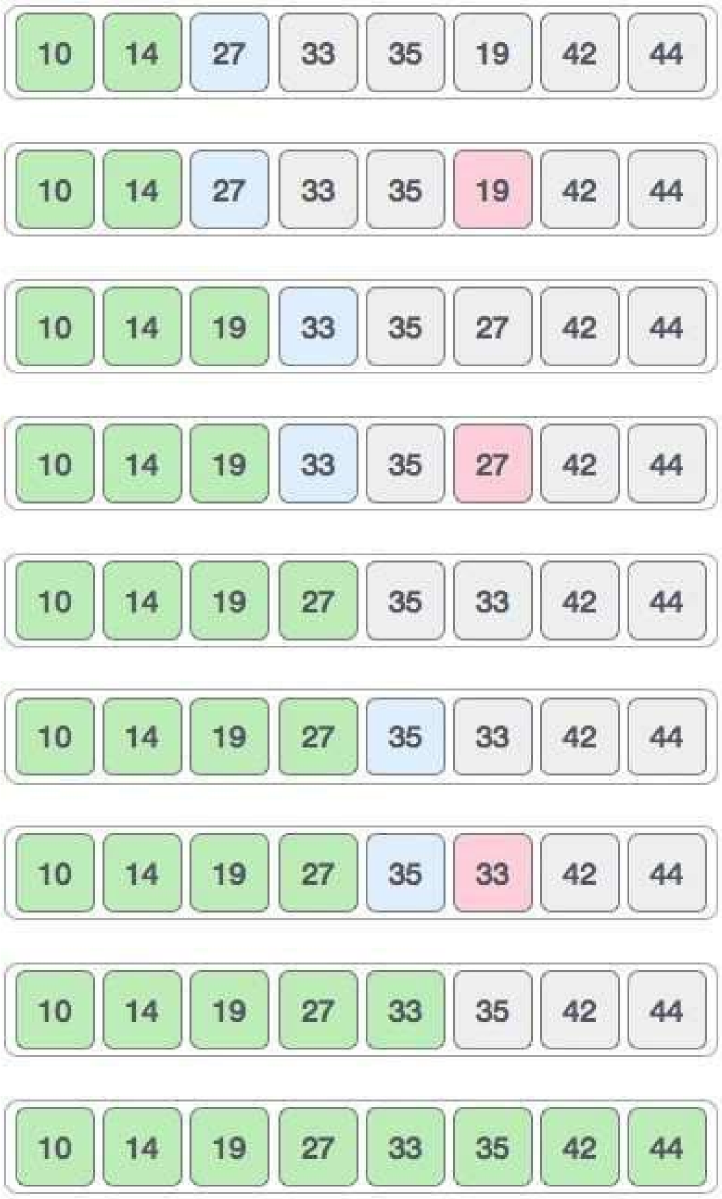


After two iterations, two least values are positioned at the beginning in a sorted manner.



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

Following is a pictorial depiction of the entire sorting process −



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
```

Selection Sort Program in C:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int
   count){ int i;

   for(i = 0;i <count-1;i++){
      printf("=");
   }

   printf("=\n");
}

void display(){
   int i;
   printf("[");

   // navigate through all
   items for(i = 0;i<MAX;i++){
      printf("%d ", intArray[i]);
   }
   printf("]\n");
}

void selectionSort(){

   int indexMin,i,j;

   // loop through all numbers
   for(i = 0; i < MAX-1; i++){

      // set    current    element    as
      minimum indexMin = i;

      // check the element to be minimum
      for(j = i+1;j<MAX;j++){
         if(intArray[j] < intArray[indexMin]){
            indexMin = j;
         }
      }

      if(indexMin != i){
         printf("Items swapped: [ %d, %d ]\n" , intArray[i],
intArray[indexMin]);

         // swap the numbers
         int temp = intArray[indexMin];
         intArray[indexMin] =
         intArray[i]; intArray[i] = temp;
      }

      printf("Iteration %d#:",(i+1));
      display();
   }
}
```

```
main(){
   printf("Input Array: ");
   display();
   printline(50);
   selectionSort();
   printf("Output Array: ");
   display();
   printline(50);
}
```

If we compile and run the above program, it will produce the following result −

```
Input Array: [4, 6, 3, 2, 1, 9, 7]
==================================================
     Items swapped: [ 4, 1 ]
iteration 1#: [1, 6, 3, 2, 4, 9, 7]
     Items swapped: [ 6, 2 ]
iteration 2#: [1, 2, 3, 6, 4, 9, 7]
iteration 3#: [1, 2, 3, 6, 4, 9, 7]
     Items swapped: [ 6, 4 ]
iteration 4#: [1, 2, 3, 4, 6, 9, 7]
iteration 5#: [1, 2, 3, 4, 6, 9, 7]
     Items swapped: [ 9, 7 ]
iteration 6#: [1, 2, 3, 4, 6, 7, 9]
Output Array: [1, 2, 3, 4, 6, 7, 9]
==================================================
```

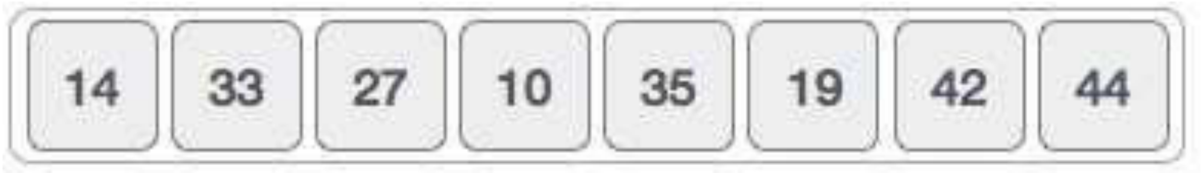
Merge sort:

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

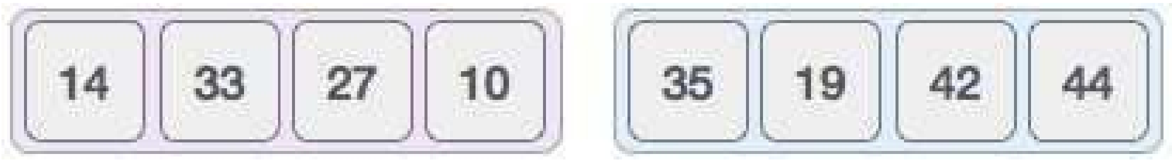
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Merge Sort Working:

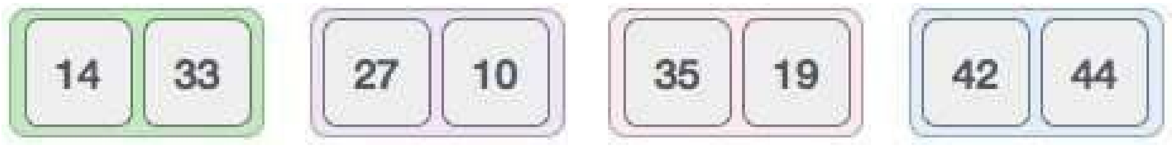
To understand merge sort, we take an unsorted array as the following −



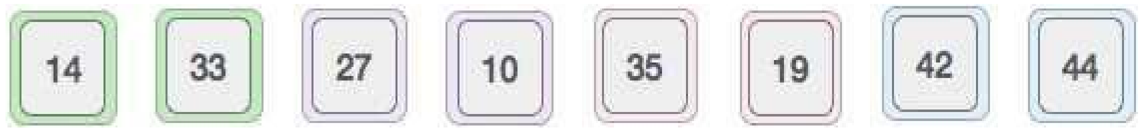
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

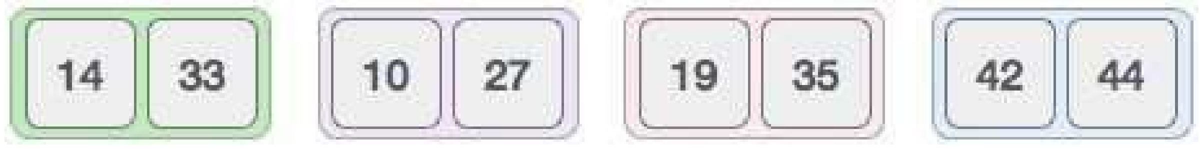


We further divide these arrays and we achieve atomic value which can no more be divided.

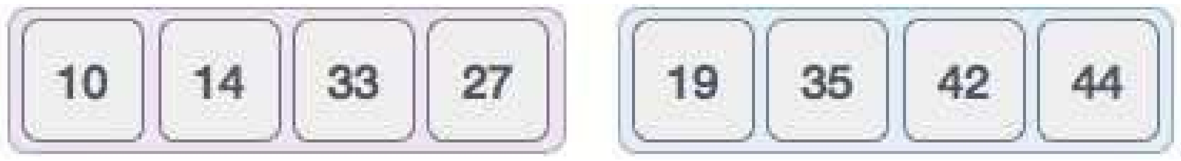


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

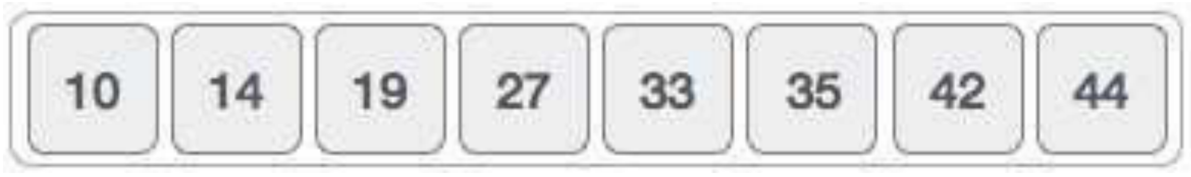
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −



Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

```
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.
```

Merge Sort Program in C:

```c
#include <stdio.h>
#define max 10

int a[10] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44 };
int b[10];



void merging(int low, int mid, int high)
   { int l1, l2, i;

   for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
      if(a[l1] <= a[l2])
         b[i] = a[l1++];
      else
         b[i] = a[l2++];
   }

   while(l1 <= mid)
      b[i++] = a[l1++];

   while(l2 <= high)
      b[i++] = a[l2++];

   for(i = low; i <= high; i++)
      a[i] = b[i];
}

void sort(int low, int high)
   { int mid;

   if(low < high) {
      mid = (low + high) / 2;
      sort(low, mid);
      sort(mid+1, high);
      merging(low, mid, high);
   }else {
      return;
   }
}

int main()
   { int i;

   printf("List before sorting\n");

   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);

   sort(0, max);

   printf("\nList after sorting\n");

   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);
}
```

If we compile and run the above program, it will produce the following result –

```
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
0 10 14 19 26 27 31 33 35 42 44
```

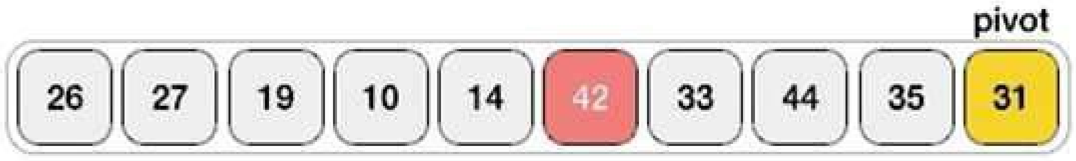
Quick sort:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large- sized data sets as its average and worst case complexity are of O(nlogn), where **n** is the number of items.

Partition in Quick Sort:

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

```
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

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows –

```
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
```

Quick Sort Program in C

```
 #include <stdio.h>
 #include<stdbool.h>
 #define MAX 7

 int intArray[MAX] = {4,6,3,2,1,9,7};

 void printline(int
    count){ int i;

    for(i = 0;i <count-
       1;i++){ printf("=");
    }

    printf("=\n");



}

void display(){
   int i;
   printf("[");

   // navigate through all
   items for(i = 0;i<MAX;i++){
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void swap(int num1, int num2){
   int temp = intArray[num1];
   intArray[num1] = intArray[num2];
   intArray[num2] = temp;
}
int partition(int left, int right, int
   pivot){ int leftPointer = left -1;
```

```
int rightPointer = right;

while(true){

   while(intArray[++leftPointer] <
      pivot){ //do nothing
   }

   while(rightPointer > 0 && intArray[--rightPointer] >
      pivot){ //do nothing
   }

   if(leftPointer >=
      rightPointer){ break;
   }else{
      printf(" item swapped :%d,%d\n",
            intArray[leftPointer],intArray[rightPointer]);
            swap(leftPointer,rightPointer);
         }

      }

      printf(" pivot swapped :%d,%d\n",
      intArray[leftPointer],intArray[right]); swap(leftPointer,right);
      printf("Updated Array: ");
      display();
      return leftPointer;
   }

   void quickSort(int left, int
      right){ if(right-left <= 0){
         return;
      }else {
         int pivot = intArray[right];
         int partitionPoint = partition(left, right, pivot);
         quickSort(left,partitionPoint-1);
         quickSort(partitionPoint+1,right);
      }
   }

   main(){
      printf("Input Array: ");
      display();
      printline(50);
      quickSort(0,MAX-1);
      printf("Output Array:
      "); display();
      printline(50);
   }
```

If we compile and run the above program, it will produce the following result −

```
Input Array: [4 6 3 2 1 9 7 ]
==================================================
 pivot swapped :9,7
Updated Array: [4 6 3 2 1 7 9 ]
 pivot swapped :4,1
Updated Array: [1 6 3 2 4 7 9 ]
 item swapped :6,2
 pivot swapped :6,4
Updated Array: [1 2 3 4 6 7 9 ]
 pivot swapped :3,3
Updated Array: [1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]
==================================================
```

References:

1. "Data Structures & Algorithm" , Tutorials Point India (https://www.tutorialspoint.com)