What is Sorting?

Sorting is a process of ordering or placing a list of elements from a collection in some kind of order. It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier.

For example, suppose we have a record of employee. It has following data:

Employee No.
Employee Name
Employee Salary
Department Name

Here, employee no. can be takes as key for sorting the records in ascending or descending order. Now, we have to search a Employee with employee no. 116, so we don't require to search the complete record, simply we can search between the Employees with employee no. 100 to 120.

Sorting Techniques

Sorting technique depends on the situation. It depends on two parameters.

1. Execution time of program that means time taken for execution of program.
2. Space that means space taken by the program.

Sorting techniques are differentiated by their efficiency and space requirements.

Sorting can be performed using several techniques or methods, as follows:

1. Bubble Sort
2. Insertion Sort

3. Selection Sort
4. Quick Sort
5. Heap Sort
6. Merge Sort
7. Shell Sort

1. Bubble Sort

- Bubble sort is a type of sorting.
- It is used for sorting 'n' (number of items) elements.
- It compares all the elements one by one and sorts them based on their values.

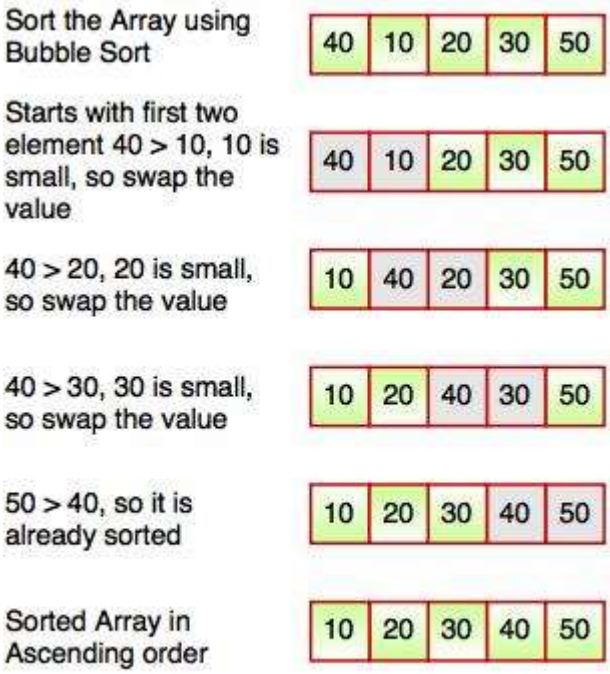


Fig. Working of Bubble Sort

- The above diagram represents how bubble sort actually works. This sort takes $O(n^2)$ time. It starts with the first two elements and sorts them in ascending order.
- Bubble sort starts with first two elements. It compares the element to check which one is greater.
- In the above diagram, element 40 is greater than 10, so these values must be swapped. This operation continues until the array is sorted in ascending order.

Example: Program for Bubble Sort

```
#include <stdio.h>
void bubble_sort(long [], long);

int main()
{
  long array[100], n, c, d, swap;
  printf("Enter Elements\n");
  scanf("%ld", &n);
  printf("Enter %ld integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%ld", &array[c]);
  bubble_sort(array, n);
  printf("Sorted list in ascending order:\n");
  for ( c = 0 ; c < n ; c++ )
     printf("%ld\n", array[c]);
  return 0;
}
void bubble_sort(long list[], long n)
{
  long c, d, t;
  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        /* Swapping */
        t        = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

Output:

```
Enter Elements
5
Enter 5 integers
20
10
40
30
50
Sorted list in ascending order:
10
20
30
40
50
```

2. Insertion Sort

- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- Insertion sort has one of the simplest implementation.
- This sort is efficient for smaller data sets but it is insufficient for larger lists.
- It has less space complexity like bubble sort.
- It requires single additional memory space.
- Insertion sort does not change the relative order of elements with equal keys because it is stable.

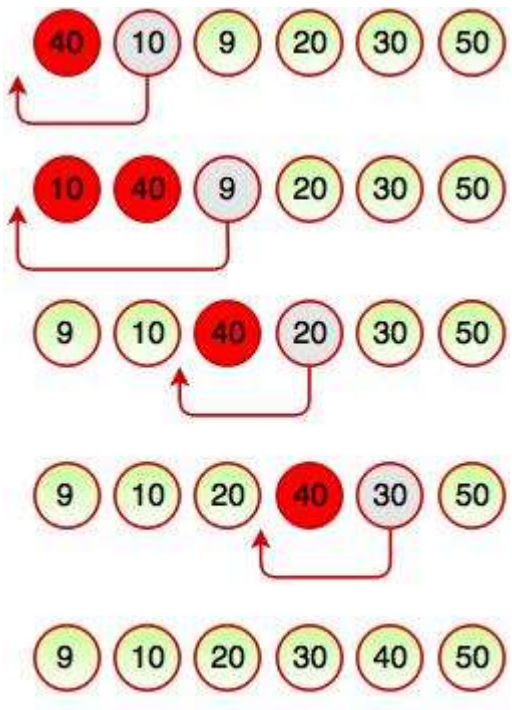


Fig. Working of Insertion Sort

- The above diagram represents how insertion sort works. Insertion sort works like the way we sort playing cards in our hands. It always starts with the second element as key. The key is compared with the elements ahead of it and is put it in the right place.
- In the above figure, 40 has nothing before it. Element 10 is compared to 40 and is inserted before 40. Element 9 is smaller than 40 and 10, so it is inserted before 10 and this operation continues until the array is sorted in ascending order.

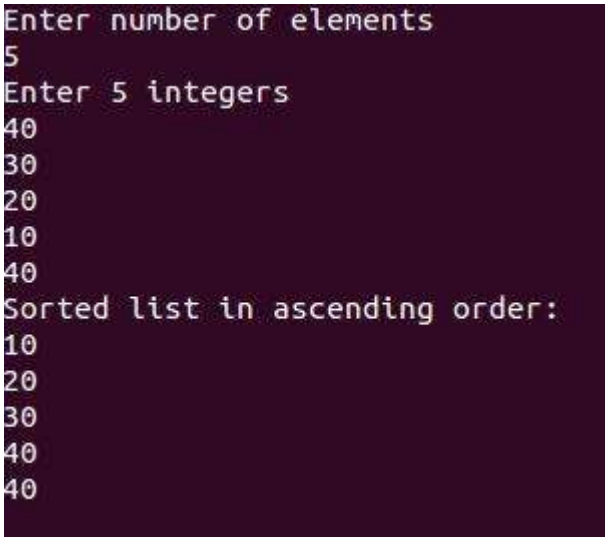
Example: Program for Insertion Sort

```
#include <stdio.h>

int main()
{
  int n, array[1000], c, d, t;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
```

```
  for (c = 0; c < n; c++)
  {
    scanf("%d", &array[c]);
  }
  for (c = 1 ; c <= n - 1; c++)
  {
    d = c;
    while ( d > 0 && array[d] < array[d-1])
    {
       t          = array[d];
       array[d]   = array[d-1];
       array[d-1] = t;
       d--;
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c <= n - 1; c++)
  {
     printf("%d\n", array[c]);
  }
  return 0;
}
```

Output:



Selection Sort

- Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position. Then finds the second smallest element and exchanges it with the element in the second position and continues until the entire array is sorted.

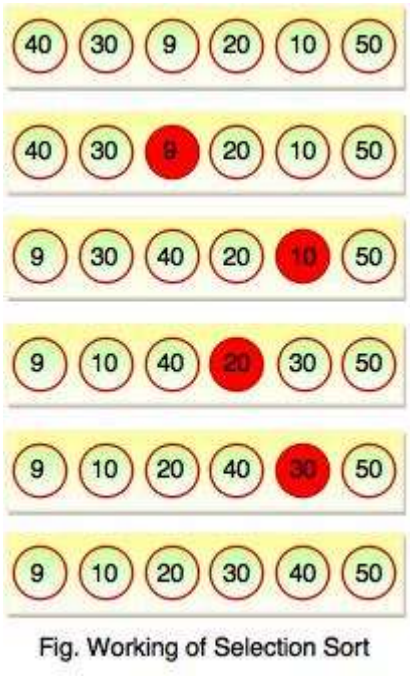


Fig. Working of Selection Sort

- In the above diagram, the smallest element is found in first pass that is 9 and it is placed at the first position. In second pass, smallest element is searched from the rest of the element excluding first element. Selection sort keeps doing this, until the array is sorted.

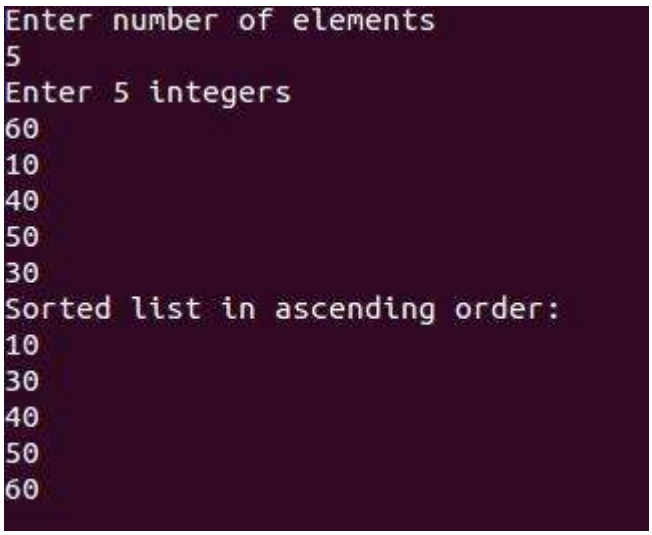
Example: Program for Selection Sort

```
#include <stdio.h>

int main()
{
  int array[100], n, c, d, position, swap;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for ( c = 0 ; c < n ; c++ )
```

```
      scanf("%d", &array[c]);
   for ( c = 0 ; c < ( n - 1 ) ; c++ )
   {
      position = c;
      for ( d = c + 1 ; d < n ; d++ )
      {
         if ( array[position] > array[d] )
            position = d;
      }
      if ( position != c )
      {
         swap = array[c];
         array[c] = array[position];
         array[position] = swap;
      }
   }
   printf("Sorted list in ascending order:\n");
   for ( c = 0 ; c < n ; c++ )
      printf("%d\n", array[c]);
   return 0;
}
```

Output:



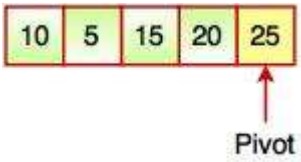
Quick Sort

- Quick sort is also known as **Partition-exchange sort** based on the rule of **Divide and Conquer.**
- It is a highly efficient sorting algorithm.
- Quick sort is the quickest comparison-based sorting algorithm.
- It is very fast and requires less additional space, only O(n log n) space is required.
- Quick sort picks an element as pivot and partitions the array around the picked pivot.

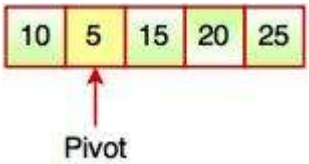
There are different versions of quick sort which choose the pivot in different ways:

1. First element as pivot

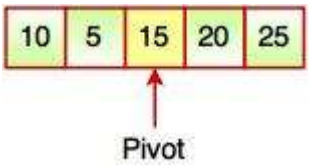
2. Last element as pivot



3. Random element as pivot



4. Median as pivot



Algorithm for Quick Sort

Step 1: Choose the highest index value as 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 index.

Step 5: While value at left < (Less than) pivot move right.

Step 6: While value at right > (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 = (Less than or Equal to) right, the point where they met is new pivot.

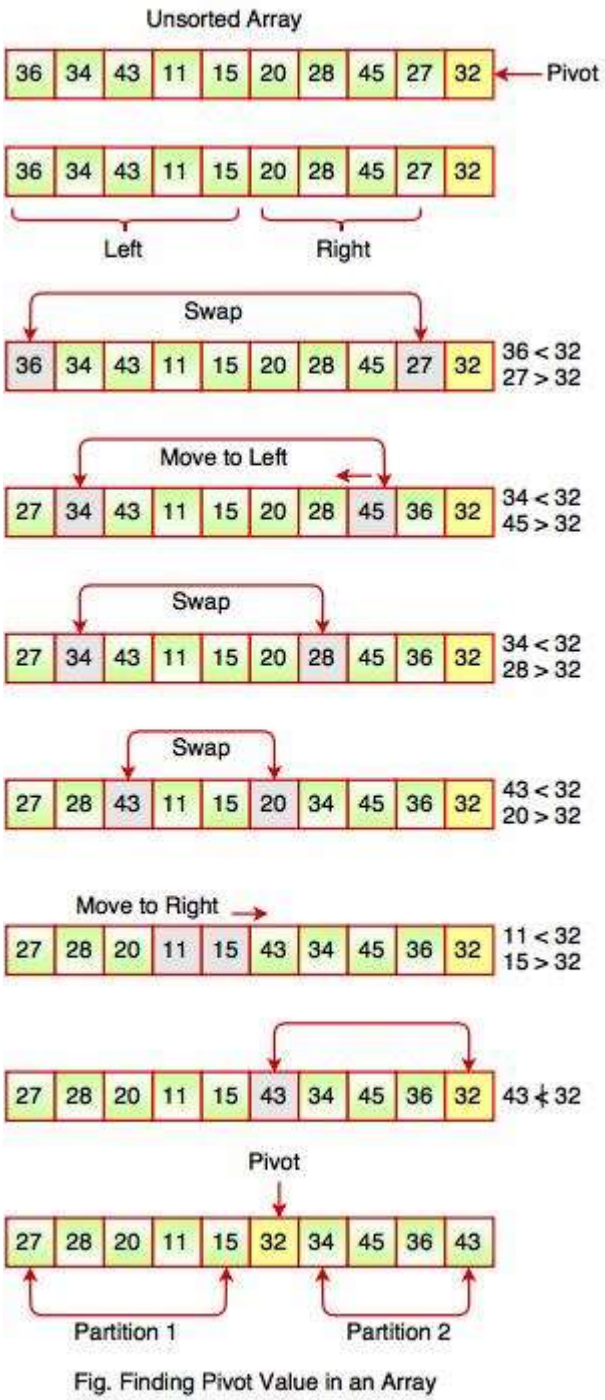


Fig. Finding Pivot Value in an Array

The above diagram represents how to find the pivot value in an array. As we see, pivot value divides the list into two parts (partitions) and then each part is processed for quick sort. Quick sort is a recursive function. We can call the partition function again.

Example: Demonstrating Quick Sort

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

//quick Sort function to Sort Integer array list
void quicksort(int array[], int firstIndex, int lastIndex)
{
   //declaaring index variables
   int pivotIndex, temp, index1, index2;
   if(firstIndex < lastIndex)
   {
      //assigninh first element index as pivot element
      pivotIndex = firstIndex;
      index1 = firstIndex;
      index2 = lastIndex;
      //Sorting in Ascending order with quick sort
      while(index1 < index2)
      {
         while(array[index1] <= array[pivotIndex] && index1 < lastIndex)
         {
            index1++;
         }
         while(array[index2]>array[pivotIndex])
         {
            index2--;
         }
         if(index1<index2)
         {
            //Swapping opertation
            temp = array[index1];
            array[index1] = array[index2];
            array[index2] = temp;
         }
      }
      //At the end of first iteration, swap pivot element with index2 element
      temp = array[pivotIndex];
```

```
        array[pivotIndex] = array[index2];
        array[index2] = temp;
        //Recursive call for quick sort, with partiontioning
        quicksort(array, firstIndex, index2-1);
        quicksort(array, index2+1, lastIndex);
    }
}
int main()
{
    //Declaring variables
    int array[100],n,i;
    //Number of elements in array form user input
    printf("Enter the number of element you want to Sort : ");
    scanf("%d",&n);
    //code to ask to enter elements from user equal to n
    printf("Enter Elements in the list : ");
    for(i = 0; i < n; i++)
    {
        scanf("%d",&array[i]);
    }
    //calling quickSort function defined above
    quicksort(array,0,n-1);
    //print sorted array
    printf("Sorted elements: ");
    for(i=0;i<n;i++)
       printf(" %d",array[i]);
    getch();
    return 0;
}
```

Output:

```
Enter the number of element you want to Sort : 5
Enter Elements in the list : 30
6
8
4
10
Sorted elements:  4 6 8 10 30
```

Heap Sort

- Heap sort is a comparison based sorting algorithm.
- It is a special tree-based data structure.
- Heap sort is similar to selection sort. The only difference is, it finds largest element and places the it at the end.
- This sort is not a stable sort. It requires a constant space for sorting a list.
- It is very fast and widely used for sorting.

It has following two properties:

1. Shape Property
2. Heap Property

1. Shape property represents all the nodes or levels of the tree are fully filled. Heap data structure is a complete binary tree.

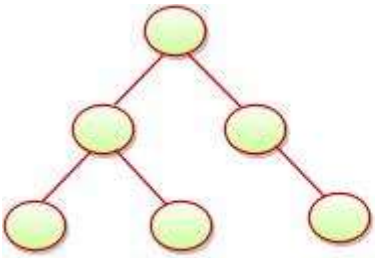


Fig. Complete-Binary Tree

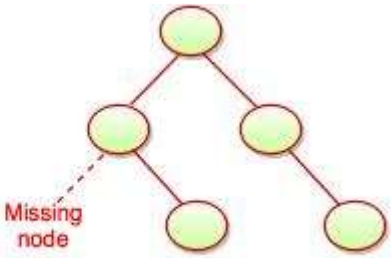


Fig. In-Complete-Binary Tree

2. Heap property is a binary tree with special characteristics. It can be classified into two types:

I. Max-Heap

II. Min Heap

I. Max Heap: If the parent nodes are greater than their child nodes, it is called a **Max-Heap.**

II. Min Heap: If the parent nodes are smaller than their child nodes, it is called a **Min-Heap.**

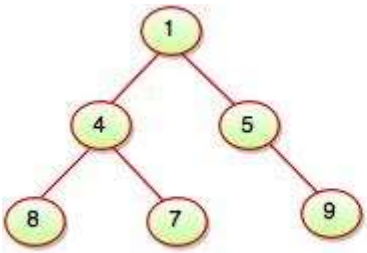


Fig. Min Heap

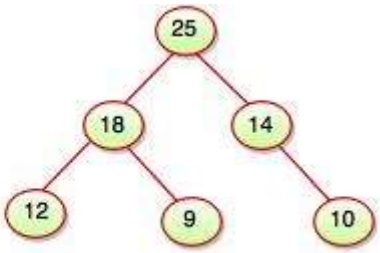


Fig. Max Heap

Example: Program for Heap Sort

```
#include <stdio.h>
void main()
{
   int heap[10], no, i, j, c, root, temp;
   printf("\n Enter no of elements :");
   scanf("%d", &no);
   printf("\n Enter the nos : ");
   for (i = 0; i < no; i++)
      scanf("%d", &heap[i]);
   for (i = 1; i < no; i++)
   {
      c = i;
      do
      {
         root = (c - 1) / 2;
         if (heap[root] < heap[c])   /* to create MAX heap array */
         {
            temp = heap[root];
            heap[root] = heap[c];
            heap[c] = temp;
```

```
         }
         c = root;
      } while (c != 0);
   }
   printf("Heap array : ");
   for (i = 0; i < no; i++)
      printf("%d\t ", heap[i]);
   for (j = no - 1; j >= 0; j--)
   {
      temp = heap[0];
      heap[0] = heap[j];    /* swap max element with rightmost leaf element
*/
      heap[j] = temp;
      root = 0;
      do
      {
         c = 2 * root + 1;    /* left node of root element */
         if ((heap[c] < heap[c + 1]) && c < j-1)
            c++;
         if (heap[root]<heap[c] && c<j)    /* again rearrange to max heap
array */
         {
            temp = heap[root];
            heap[root] = heap[c];
            heap[c] = temp;
         }
         root = c;
      } while (c < j);
   }
   printf("\n The sorted array is : ");
   for (i = 0; i < no; i++)
     printf("\t %d", heap[i]);
   printf("\n Complexity : \n Best case = Avg case = Worst case = O(n logn)
\n");
}
```

Output:

```
 Enter no of elements :5

 Enter the nos : 10
6
30
9
40
Heap array : 40  30      10      6       9
 The sorted array is :   6       9       10      30      40
 Complexity :
 Best case = Avg case = Worst case = O(n logn)
```

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.

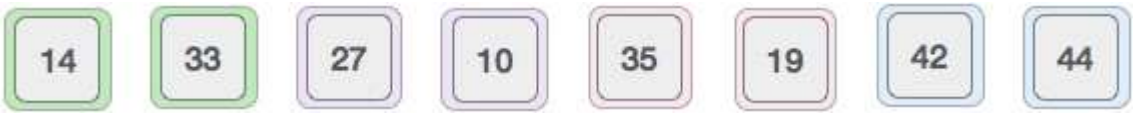
How Merge Sort Works?

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 −

Now we should learn some programming aspects of merge sorting.

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.

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

```
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
```

SHELL Sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula as −

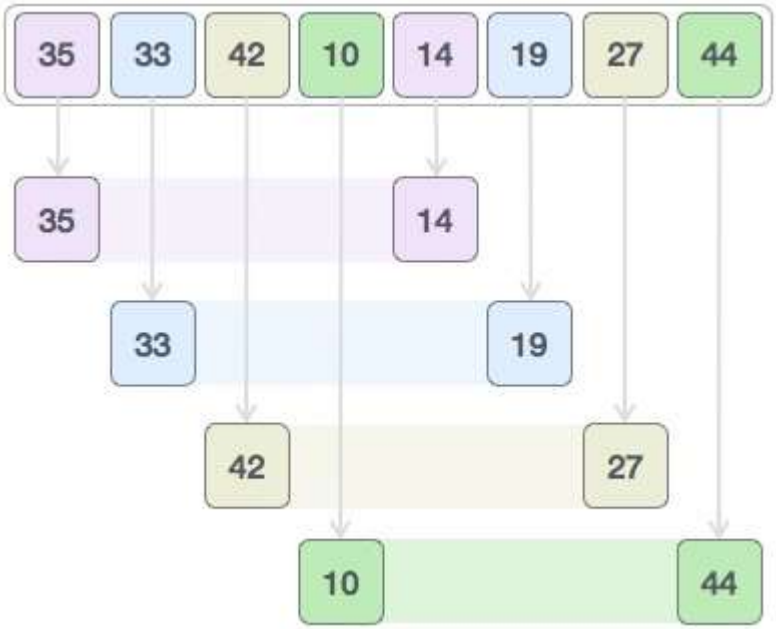
Knuth's Formula

```
h = h * 3 + 1
where –
   h is interval with initial value 1
```

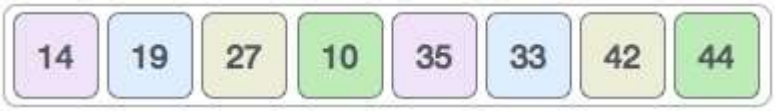
This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).

How Shell Sort Works?

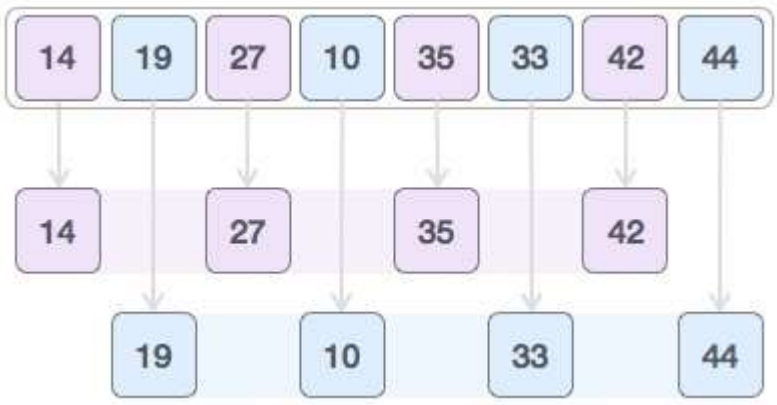
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



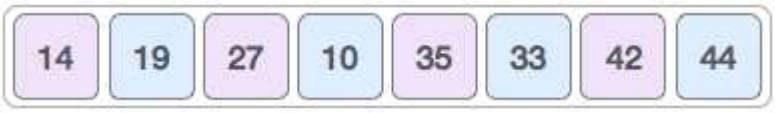
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −

We see that it required only four swaps to sort the rest of the array.

Algorithm

Following is the algorithm for shell sort.

Step 1 − Initialize the value of *h*
Step 2 − Divide the list into smaller sub-list of equal interval *h*
Step 3 − Sort these sub-lists using **insertion sort**
Step 3 − Repeat until complete list is sorted

Pseudocode

Following is the pseudocode for shell sort.

```
procedure shellSort()
   A : array of items

   /* calculate interval*/
   while interval < A.length /3 do:
      interval = interval * 3 + 1
   end while

   while interval > 0 do:

      for outer = interval; outer < A.length; outer ++ do:

      /* select value to be inserted */
      valueToInsert = A[outer]
      inner = outer;

         /*shift element towards right*/
         while inner > interval -1 && A[inner - interval] >=
valueToInsert do:
            A[inner] = A[inner - interval]
            inner = inner - interval
         end while

      /* insert the number at hole position */
      A[inner] = valueToInsert

      end for

   /* calculate interval*/
   interval = (interval -1) /3;

   end while

end procedure
```