Unit-2

Sorting and Searching

What is Sorting – Definition

- Sorting is the **process of arranging data into meaningful order** so that you can analyze it more effectively.

- Often in real life, we are supposed to arrange data in a particular order. For instance, during our school days, we are told to stand in the queue based on our heights. Another example is the attendance register at school/college which contains our names arranged in alphabetical order.

- These data arrangements give easier access to data for future use for ex. finding "Joe" in an attendance register of 100 students.

- **The arrangement of data in a particular order is called as sorting of the data by that order.**

- Two of the most commonly used orders are:
  - Ascending order
  - Descending order

Ascending order : the data in ascending order, we try to arrange the data in a way such that each element is in some way **"smaller than"** its successor.
As a simple example, the numbers 1, 2, 3, 4, 5 are sorted in ascending order. Here, the "smaller than" relation is actually the "<" operator. As can be seen, 1 < 2 < 3 < 4 < 5.

Descending order: descending order is the exact opposite of ascending order. Given a data that is sorted in ascending order, reverse it and you will get the data in descending order.

Types of Sorting

1. **Bubble Sorting**
2. **Insertion Sorting**
3. **Quick Sorting**
4. **Merge Sorting**
5. **Selection Sorting**
6. **Shell Sorting**
7. **Bucket Sorting**

Sort Algorithm Introduction

1. Bubble Sort

Bubble Sort is a **sorting algorithm** where we repeatedly iterate through the array and swap adjacent elements that are unordered. We repeat this until the array is sorted.

- **Bubble sort is one of the easiest sorting techniques in programming and it is very simple to implement.**

- **It just simply compares the current element with the next element and swaps it, if it is greater or less, depending on the condition.**

- **It gives quite accurate results. Each time an element is compared with all other elements till it's final place is found is called a pass.**

- **Bubble sort gets its name because it filters out the elements at the top of the array like bubbles on water.**

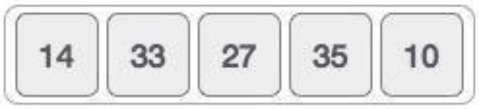
- Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world.
- **It is not suitable for large data sets.** The **average** and **worst-case complexity** of Bubble sort is **$O(n^2)$,** where **n** is a number of items.

Bubble short is majorly used where –

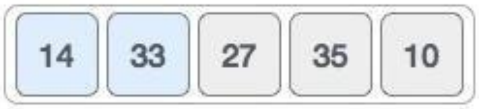
- complexity does not matter
- simple and short code is preferred

How Bubble Sort Works?

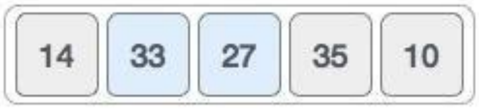
We take an unsorted array for our example.



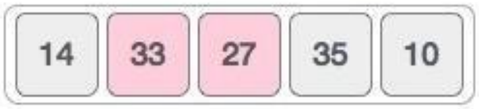
Bubble sort starts with very first two elements, comparing them to check which one is greater.



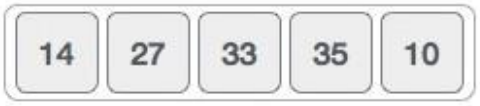
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



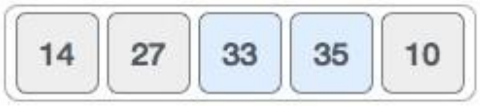
We find that 27 is smaller than 33 and these two values must be swapped.



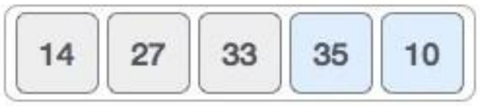
The new array should look like this −



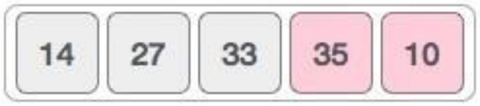
Next we compare 33 and 35. We find that both are in already sorted positions.



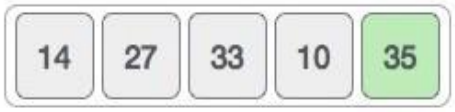
Then we move to the next two values, 35 and 10.



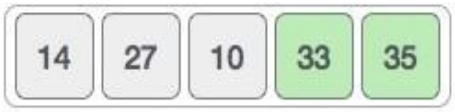
We know then that 10 is smaller 35. Hence they are not sorted.



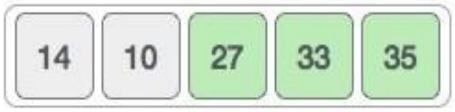
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



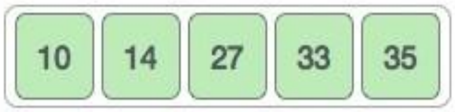
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



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.



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

Bubble sort algorithm

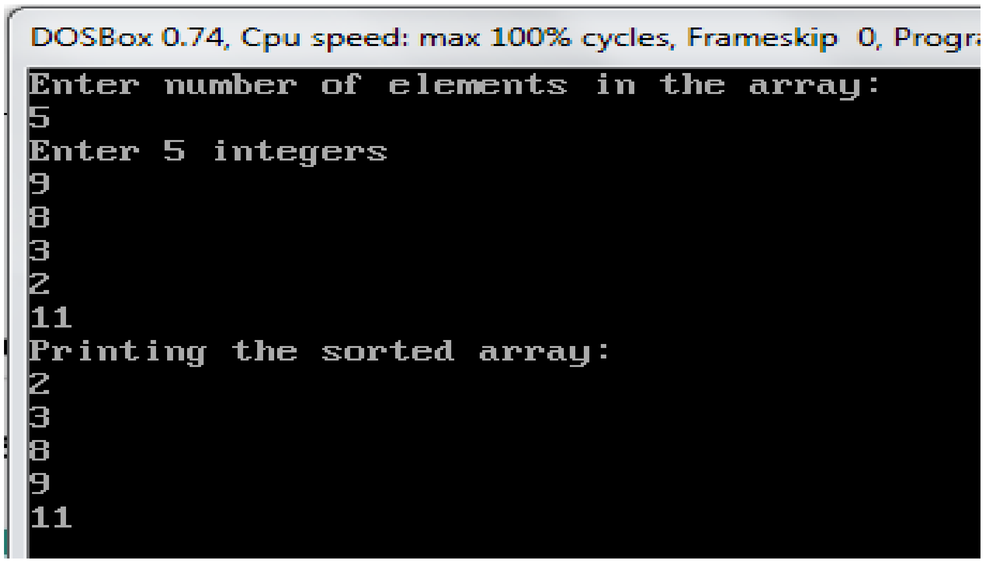
1: Start comparing each element with its adjacent element from the starting index.
2: If the current and the next element are out of order, swap them.
3: Repeat step 2 for all the elements of the array/list.
4: Repeat steps 1, 2, and 3 until we have reached the final sorted state of the array.

Example

```
#include <stdio.h>
void bubble_sort(int a[], int n)
{
   int i = 0, j = 0, temp;
   for (i = 0; i < n; i++)
   {   // loop n times - 1 per element
      for (j = 0; j < n - i - 1; j++)
      { // last i elements are sorted already
         if (a[j] > a[j + 1])
         {  // swop if order is broken
            temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
         }
      }
   }
}
```

```
int main()
{
  int a[100], n, i;
  printf("Enter number of elements in the array:\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
  bubble_sort(a, n);
  printf("Printing the sorted array:\n");
  for (i = 0; i < n; i++)
     printf("%d\n", a[i]);
  getch();
  return 0;
}
```

Output



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Progr

```
Enter number of elements in the array:
5
Enter 5 integers
9
8
3
2
11
Printing the sorted array:
2
3
8
9
11
```

Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1. Time Complexity

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

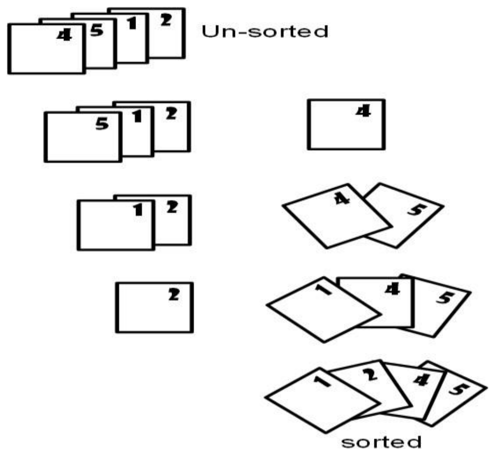
2. Space Complexity

- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.

2. Insertion Sorting

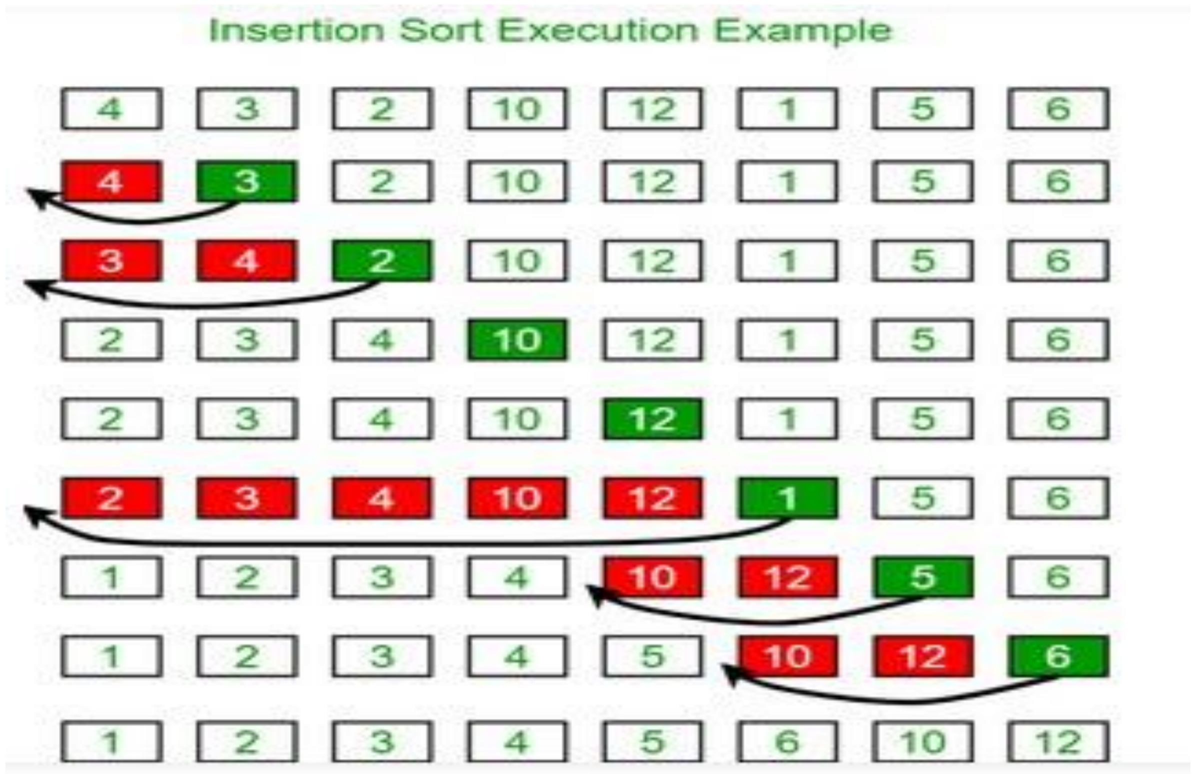
- **What is Insertion Sorting ???**

- **Insertion Sort in C** is a simple and efficient **sorting** algorithm, that creates the final **sorted** array one element at a time. It is usually implemented when the user has a small data set.



- Insertion Sort is a sorting algorithm where the array is sorted by taking one element at a time. The principle behind insertion sort is to take one element, iterate through the sorted array & find its correct position in the sorted array.

  - Insertion Sort works in a similar manner as we arrange a deck of cards.



Another Example:

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

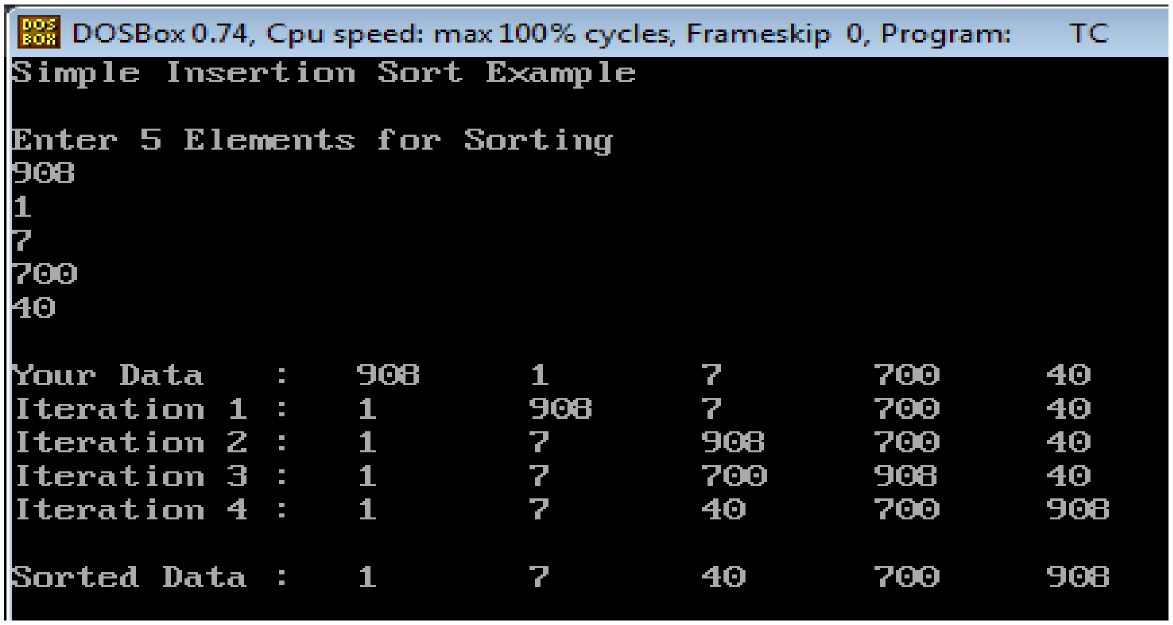
Step 6 - Repeat until the array is sorted.

Example

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

#define MAX_SIZE 5

void insertion(int[]);

int main()
{
   int arr_sort[MAX_SIZE], i;
   printf("Simple Insertion Sort Example \n");

   printf("\nEnter %d Elements for Sorting\n", MAX_SIZE);

   for (i = 0; i < MAX_SIZE; i++)
```

```
    {
        scanf("%d", &arr_sort[i]);
    }
    printf("\nYour Data   :");
    for (i = 0; i < MAX_SIZE; i++)
    {
        printf("\t%d", arr_sort[i]);
    }
    insertion(arr_sort);
    getch();
    return 0;
}
void insertion(int arr[])
{
    int i, j, a, t;
    for (i = 1; i < MAX_SIZE; i++)
    {
        t = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > t)
        {
                      arr[j + 1] = arr[j];
                        j--;
        }
        arr[j + 1] = t;
        printf("\nIteration %d : ", i);
        for (a = 0; a < MAX_SIZE; a++)
         {
             printf("\t%d", arr[a]);
         }
    }
    printf("\n\nSorted Data :");
    for (i = 0; i < MAX_SIZE; i++)
    {
        printf("\t%d", arr[i]);
    }
}
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Simple Insertion Sort Example

Enter 5 Elements for Sorting
908
1
7
700
40

Your Data   :   908     1       7       700     40
Iteration 1 :   1       908     7       700     40
Iteration 2 :   1       7       908     700     40
Iteration 3 :   1       7       700     908     40
Iteration 4 :   1       7       40      700     908

Sorted Data :   1       7       40      700     908
```

- **Uses:** Insertion sort is used when number of elements is small.
- It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

Expected Input and Output

For insertion sort algorithm, we can have the following 3 different sets of input and output.

1. Average case (Unsorted array): When the input array has random distribution of numbers.

For example:

If the input array is {4, 6, 1, 2, 5, 3}
the expected output array will have data as
{1, 2, 3, 4, 5, 6}

2. Best case (Sorted Array): When the input array is already sorted, in that case we have to make minimum number of swaps.

For example:

If the input array has data as {-3, 31, 66}
then the expected output array will have data as
{-3, 31, 66}

<u>3. Worst Case (Reverse sorted array):</u> When the array is sorted in a reverse manner, in that case we have to make maximum number of swaps.

<u>For example:</u>

If the input array has elements as {9, 8, 6, 3, 1}
then the output array will have data as {1, 3, 6, 8, 9}

3. Selection Sort

In selection sort, **the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.**

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

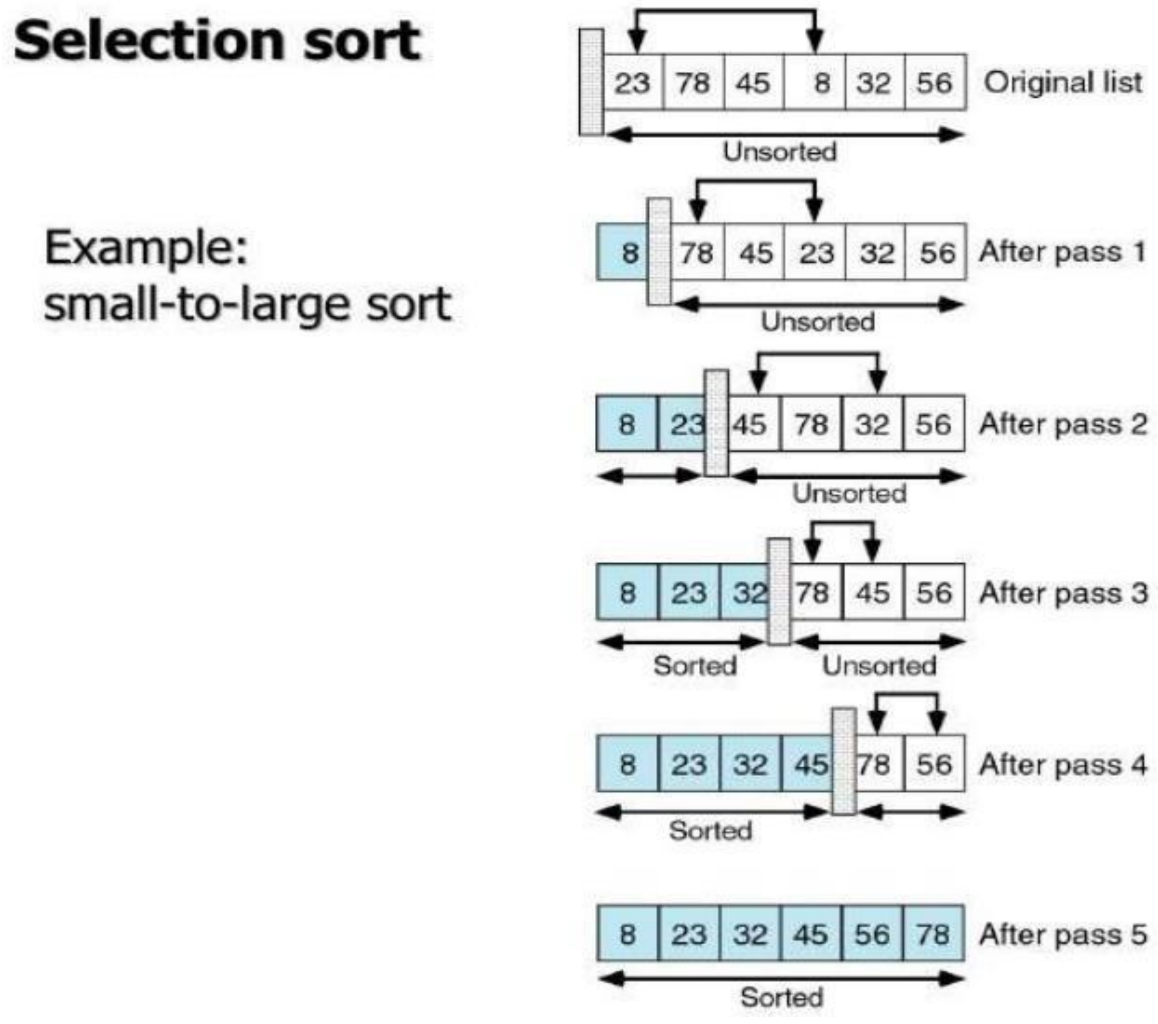
Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.

Initially, the **sorted part is empty** and the **unsorted part is the entire list.**

The algorithm **maintains two subarrays** in a given array.

- **The subarray which is already sorted.**
- **Remaining subarray which is unsorted.**

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.



Selection sort
Example:
small-to-large sort
23 78 45 8 32 56 Original list
Unsorted
8 78 45 23 32 56 After pass 1
Unsorted
8 23 45 78 32 56 After pass 2
Unsorted
8 23 32 78 45 56 After pass 3
Sorted
Unsorted
8 23 32 45 78 56 After pass 4
Sorted
8 23 32 45 56 78 After pass 5
Sorted

Step by Step Process

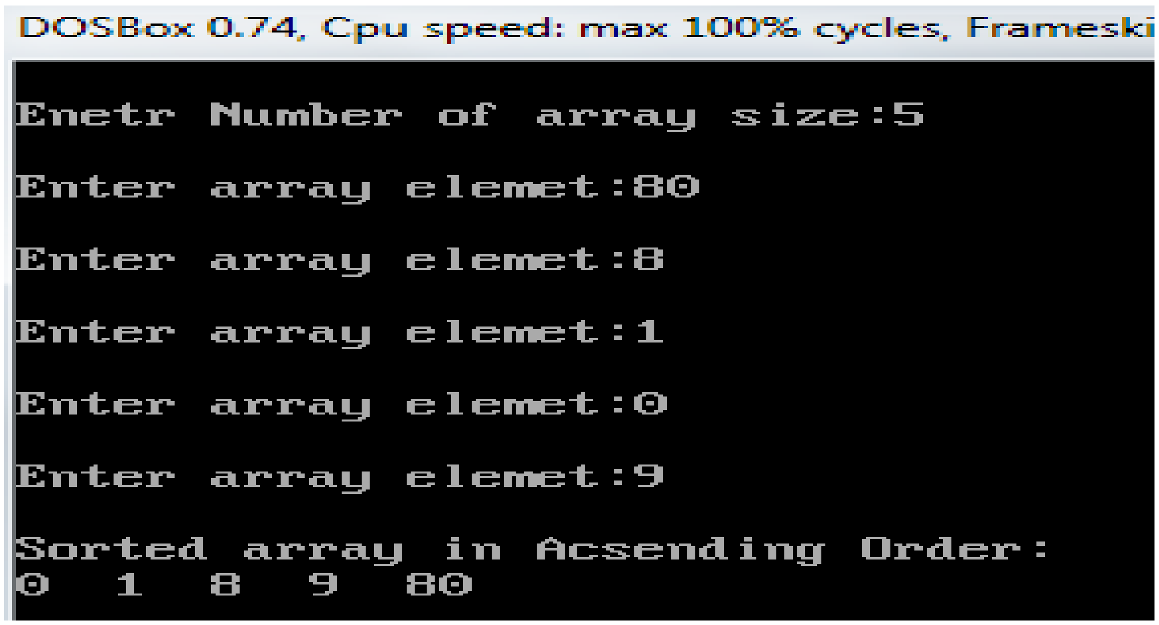
The selection sort algorithm is performed using the following steps...

- **Step 1** - Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparision, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Example :

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

// function to swap the the position of two elements
void swap(int *a, int *b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
 }

void selectionSort(int a[], int n)
{
  int i,j;
  for (i = 0; i < n - 1; i++)
  {
          int min_idx = i;
           for (j = i + 1; j < n; j++)
           {
             if (a[j] < a[min_idx])
               min_idx = j;
           }
          swap(&a[min_idx], &a[i]);
  }

 }

void printArray(int array[], int n)
{
          int i;
          for (i = 0; i < n; ++i)
          {
            printf("%d  ", array[i]);
          }
          printf("\n");
}

int main()
{
  int a[100],n,i;
  printf("\nEnetr Number of array size:");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
     printf("\nEnter array elemet:");
     scanf("%d",&a[i]);
  }
  selectionSort(a, n);
  printf("\n Sorted array in Acsending Order:\n");
  printArray(a, n);
  getch();
  return 0;
}
```

Output :



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameski

Enetr Number of array size:5

Enter array elemet:80

Enter array elemet:8

Enter array elemet:1

Enter array elemet:0

Enter array elemet:9

Sorted array in Acsending Order:
0   1   8   9   80
```

4. Shell Sort

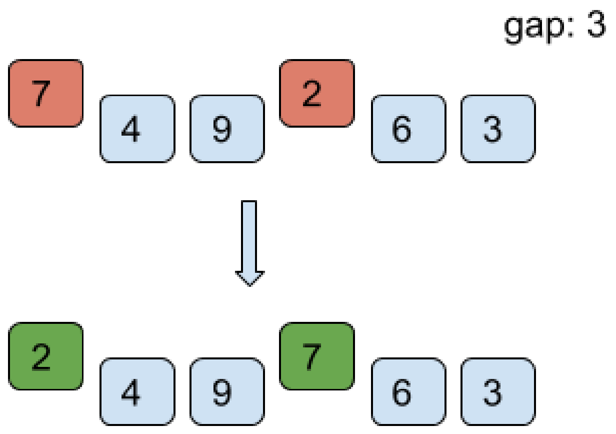
Shellsort, also known as **Shell sort or Shell's method**, is an **in-place comparison sort.**

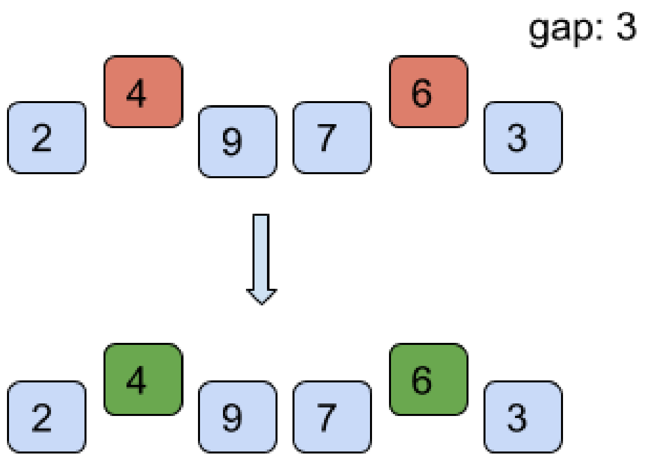
It can either be seen as a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort).
The method starts by sorting elements far apart from each other and progressively reducing the gap between them. Starting with far apart elements can move some out-of-place elements into position faster than a simple nearest neighbor exchange.

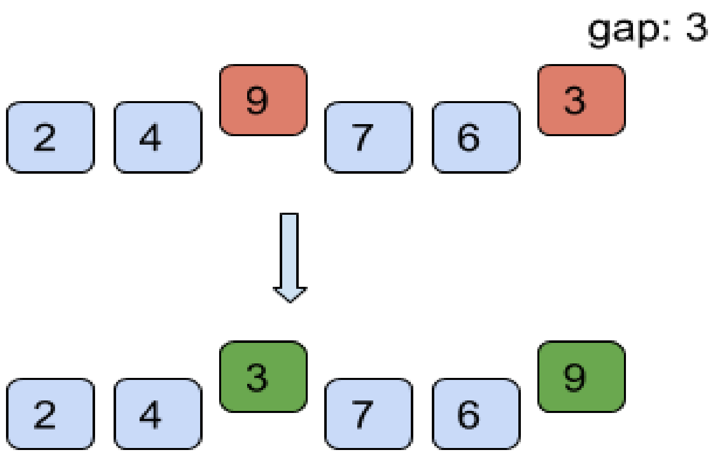
Algorithm of Shell Sort

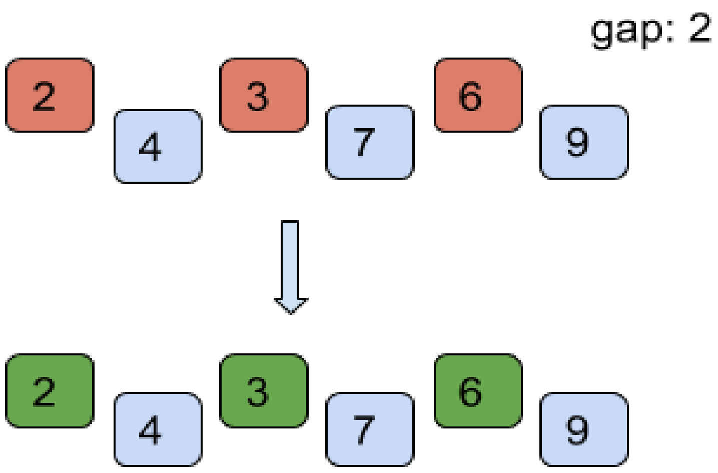
1. Initialize the gap size.
2. Divide the array into subarrays of equal gap size.
3. Apply insertion sort on the subarrays.
4. Repeat the above steps until the gap size becomes 0 resulting into a sorted array.

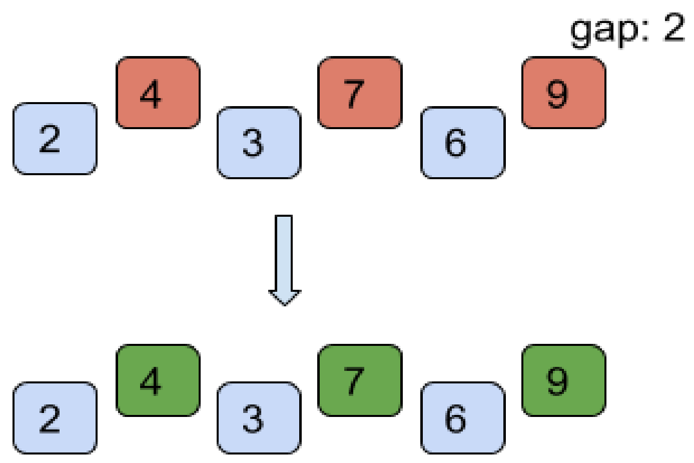
Find below the working of shell sort for the example list = [7, 4, 9, 2, 6, 3] with starting gap size = 3 and reducing the gap size by 1 after every iteration till the gap size is greater than 0.

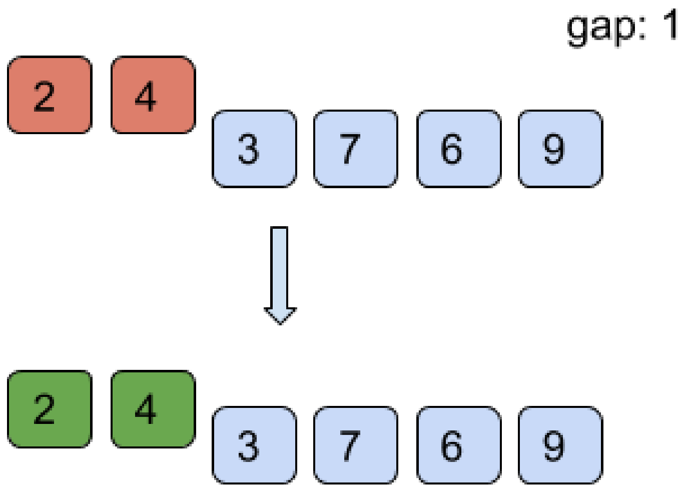


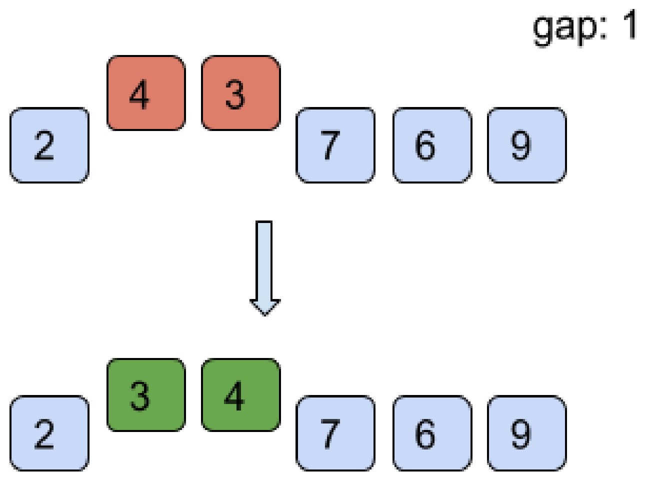

gap: 3
2 4 9 7 6 3
2 4 9 7 6 3

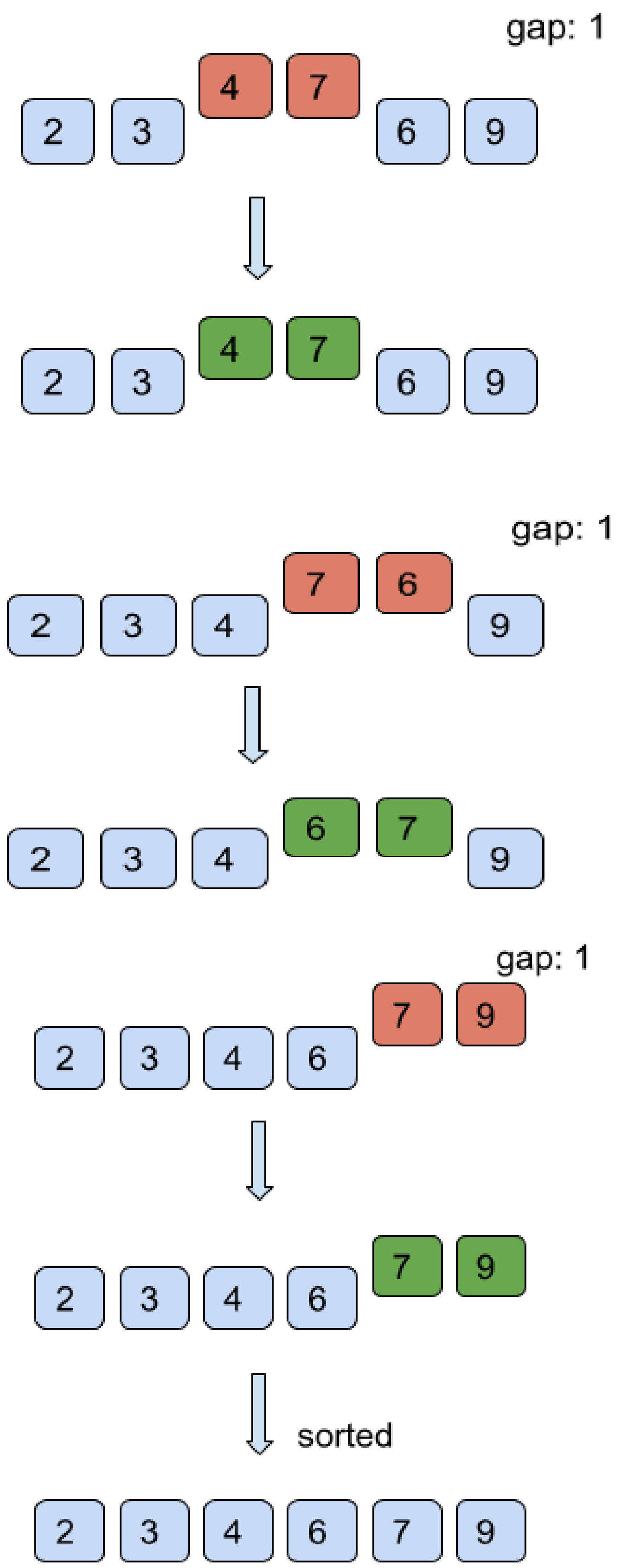

gap: 3
2 4 9 7 6 3
2 4 3 7 6 9


gap: 2
2 4 3 7 6 9
2 4 3 7 6 9


gap: 2
2 4 3 7 6 9
2 4 3 7 6 9


gap: 1
2 4 3 7 6 9
2 4 3 7 6 9

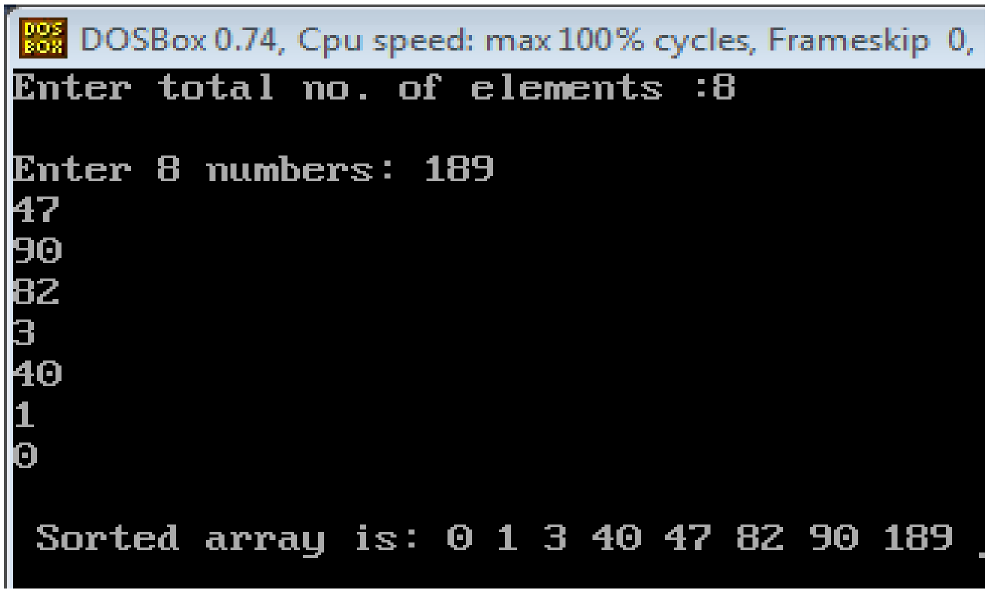

gap: 1
2 4 3 7 6 9
2 3 4 7 6 9


gap: 1
2 3 4 7 6 9
2 3 4 7 6 9
gap: 1
2 3 4 7 6 9
2 3 4 6 7 9
gap: 1
2 3 4 6 7 9
2 3 4 6 7 9
sorted
2 3 4 6 7 9

Example:

```
#include <stdio.h>
void shellsort(int arr[], int num)
{
   int i, j, k, tmp;
   for (i = num / 2; i > 0; i = i / 2)
   {
      for (j = i; j < num; j++)
      {
         for(k = j - i; k >= 0; k = k - i)
         {
            if (arr[k+i] >= arr[k])
               break;
            else
            {
               tmp = arr[k];
               arr[k] = arr[k+i];
               arr[k+i] = tmp;
            }
         }
      }
   }
}

int main()
{
   int arr[30];
   int k,  num;
   printf("Enter total no. of elements : ");
   scanf("%d", &num);
   printf("\nEnter %d numbers: ", num);

   for (k =  0 ; k < num; k++)
   {
      scanf("%d", &arr[k]);
   }
   shellsort(arr, num);
   printf("\n Sorted array is: ");
   for (k = 0; k < num; k++)
     printf("%d ", arr[k]);
   return 0;
}
```

Output :



```
Enter total no. of elements :8

Enter 8 numbers: 189
47
90
82
3
40
1
0

 Sorted array is: 0 1 3 40 47 82 90 189
```

5. Merge Sort

Introduction:

- ⦿Merge Sort is a complex and fast sorting algorithm that repeatedly divides an unsorted section into two equal subsections, sorts them separately and merges them correctly.

Definition:

- ⦿Merge sort is a **DIVIDE AND CONQUER** algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.

OR

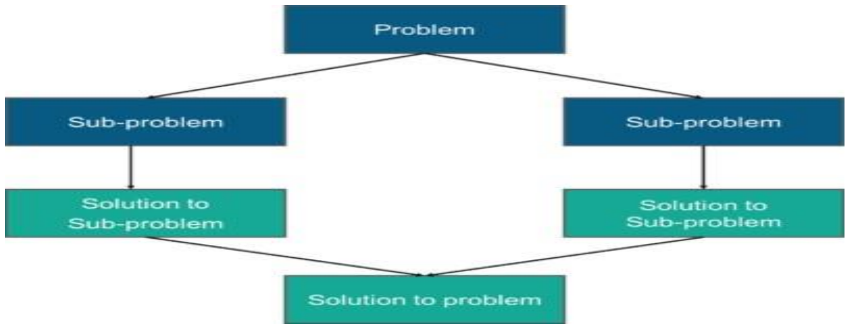
- ⦿we divide a problem into sub problems. When the solution to each sub problem is ready, we 'combine' the results from the sub problems to solve the main problem.

Steps involved:

- ⦿ ***Divide the problem into sub-problems that*** are similar to the original but smaller in size.

- ⦿ ***Conquer the sub-problems by solving them*** recursively. If they are small enough, just solve them in a straightforward manner.

⦿ ***Combine the solutions to create a solution to*** the original problem.



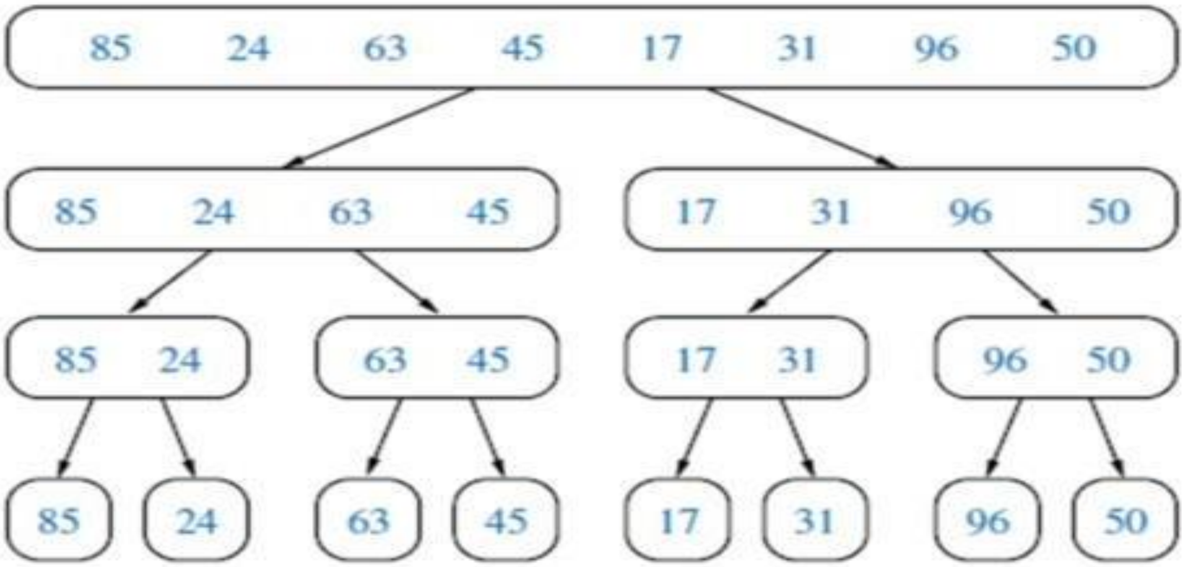
Merge Sort Algorithm

⦿ **MergeSort(arr[], l, r),** where l is the index of the first element & r is the index of the last element.

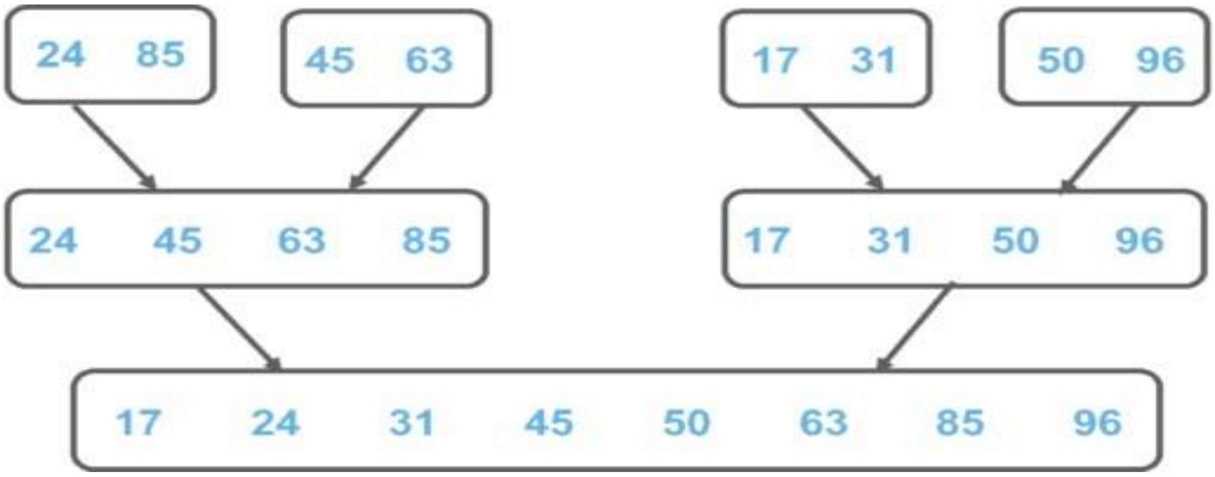
```
If l<r
1. Find the middle index of the array to divide it in two halves:
m = (l+r)/2
2. Call MergeSort for first half:
mergeSort(array, l, m)
3. Call mergeSort for second half:
mergeSort(array, m+1, r)
4. Recursively, merge the two halves in a sorted manner, so that only one sorted array is left:
merge(array, l, m, r)
```

Example:

⦿ Divide the unsorted array recursively until 1 element in each sub-array remains.



- Recursively, merge sub-arrays to produce sorted sub-arrays until all the sub-array merges and only one array remains.

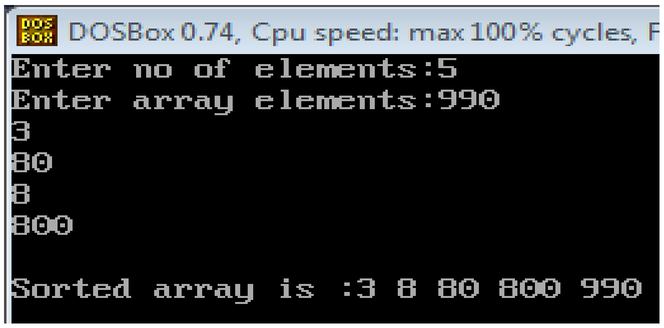


Example

```
#include<stdio.h>

void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
```

```
int main()
{
        int a[30],n,i;
        printf("Enter no of elements:");
        scanf("%d",&n);
        printf("Enter array elements:");

        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        mergesort(a,0,n-1);
        printf("\nSorted array is :");
        for(i=0;i<n;i++)
                printf("%d ",a[i]);
        getch();
        return 0;
}

void mergesort(int a[],int i,int j)
{
        int mid;

        if(i<j)
        {
                mid=(i+j)/2;
                mergesort(a,i,mid);        //left recursion
                mergesort(a,mid+1,j); //right recursion
                merge(a,i,mid,mid+1,j); //merging of two sorted sub-
arrays
        }
}

void merge(int a[],int i1,int j1,int i2,int j2)
{
        int temp[50];   //array used for merging
        int i,j,k;
        i=i1; //beginning of the first list
        j=i2; //beginning of the second list
        k=0;

        while(i<=j1 && j<=j2)      //while elements in both lists
        {
                if(a[i]<a[j])
                {
                        temp[k]=a[i];
                        i++;
                }
                else
                {
                        temp[k]=a[j];
                        j++;
                }
                k++;
        }

        while(i<=j1)
        {       //copy remaining elements of the first list
                temp[k]=a[i];
                i++;
                k++;
        }

        while(j<=j2)
        {       //copy remaining elements of the second list
                temp[k]=a[j];
                j++;
                k++;
        }

        //Transfer elements from temp[] back to a[]
        for(i=i1,j=0;i<=j2;i++,j++)
        {
                a[i]=temp[j];
        }
}
```

<u>Output:</u>



```
DOSBox 0.74, Cpu speed: max 100% cycles, F
Enter no of elements:5
Enter array elements:990
3
80
8
800

Sorted array is :3 8 80 800 990
```

6. Quicksort

Quicksort is a divide-and-conquer algorithm. It works by selecting a **'pivot'** element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

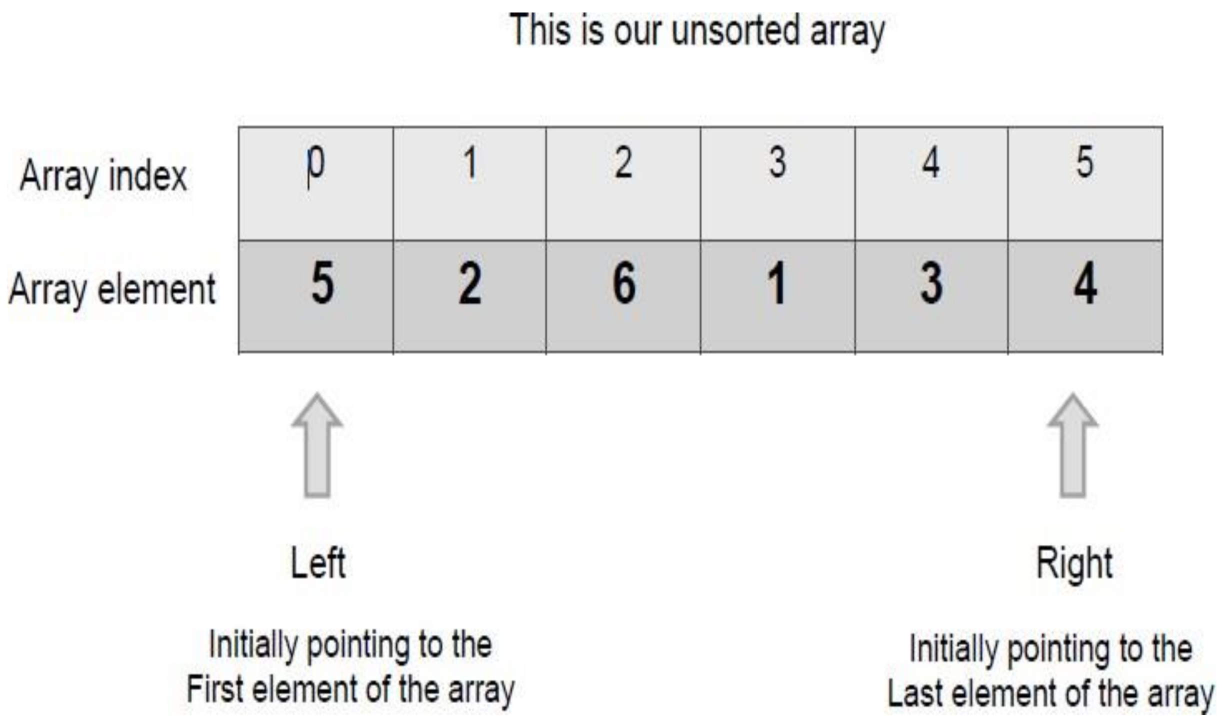
- The **key process in quickSort** is **partition().**
- Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put **all smaller elements (smaller than x) before x,** and put **all greater elements (greater than x) after x.**

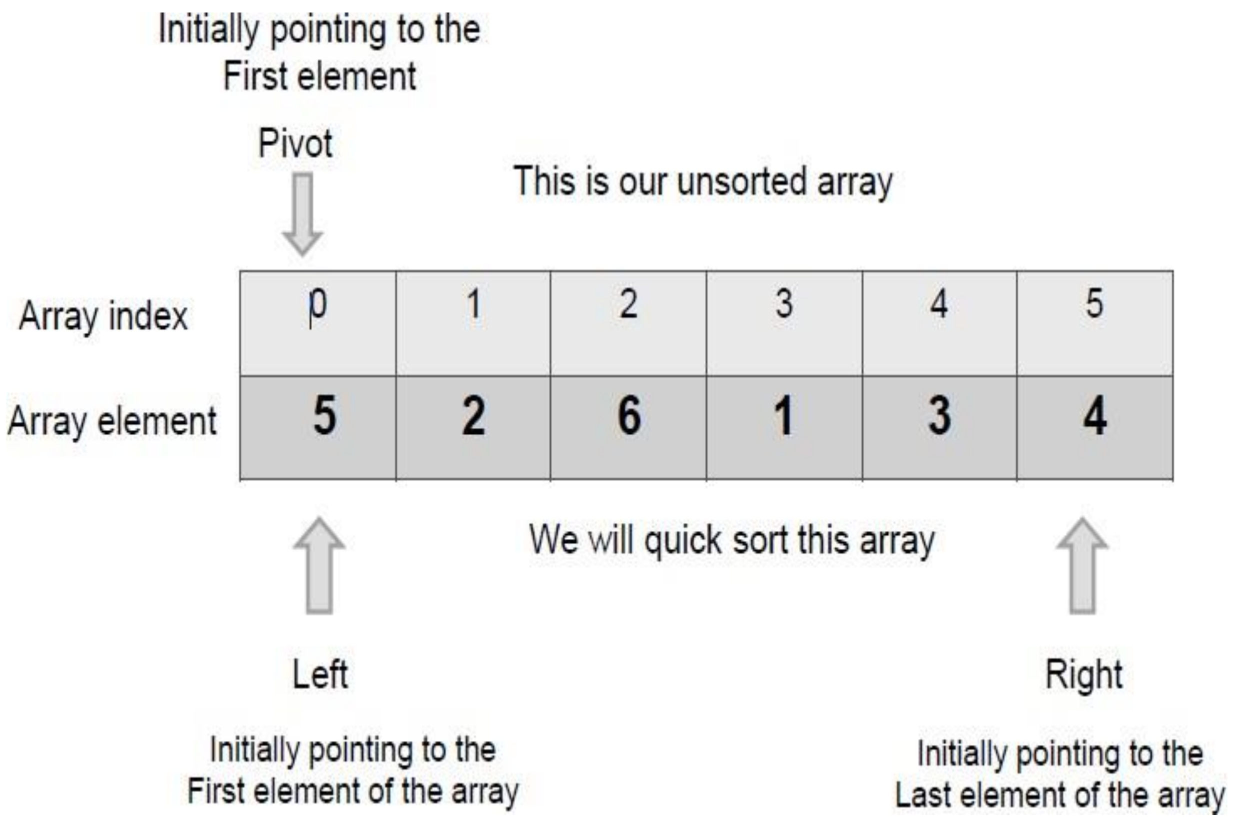
Example

Consider an array having 6 elements

5 2 6 1 3 4

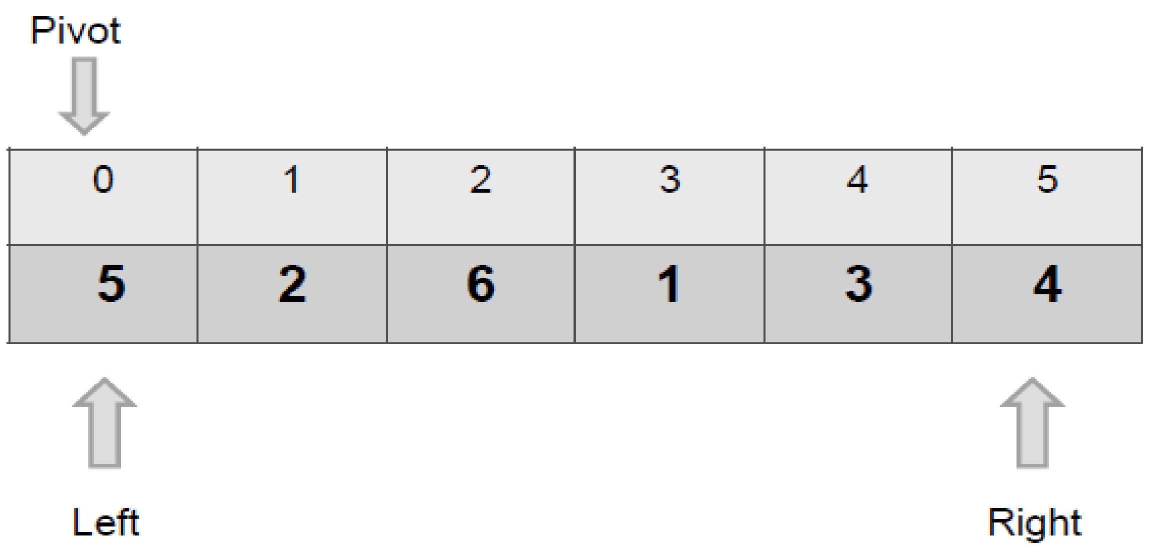
Arrange the elements in ascending order using quick sort algorithm

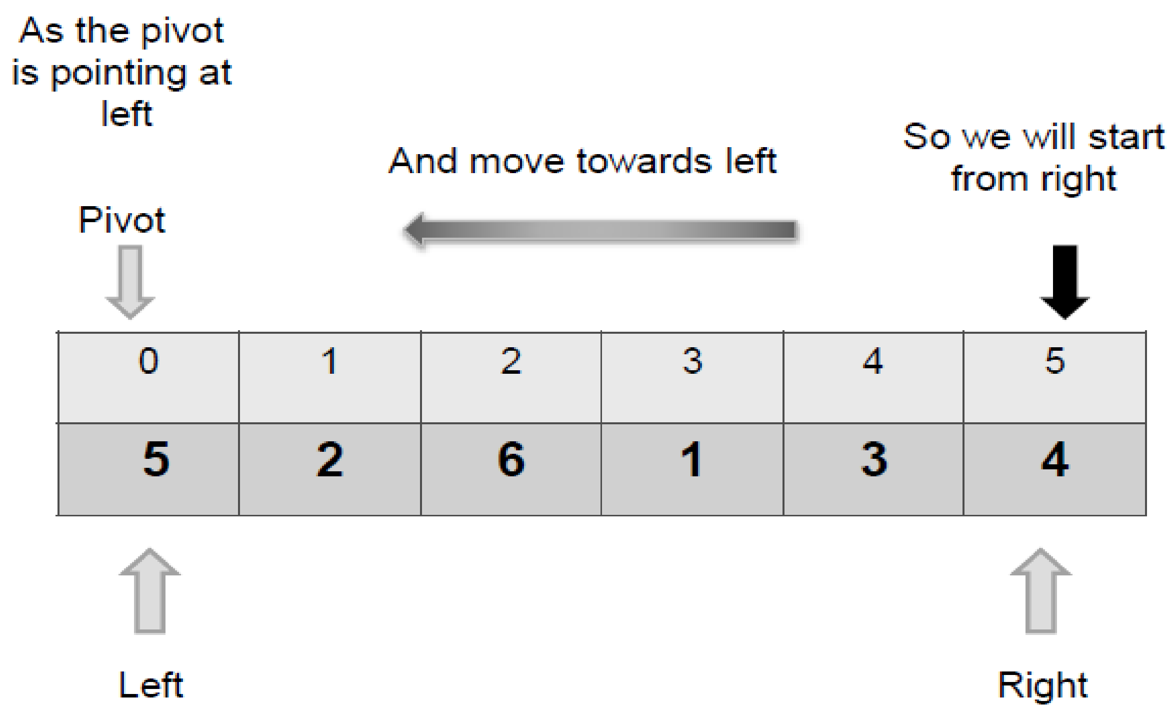

This is our unsorted array
Array index
Array element
0 1 2 3 4 5
5 2 6 1 3 4
Left
Initially pointing to the First element of the array
Right
Initially pointing to the Last element of the array


Initially pointing to the First element
Pivot
This is our unsorted array
Array index
Array element
0 1 2 3 4 5
5 2 6 1 3 4
We will quick sort this array
Left
Initially pointing to the First element of the array
Right
Initially pointing to the Last element of the array

Remember this rule:

All element to the **RIGHT** of pivot be **GREATER** than pivot.
All element to the **LEFT** of pivot be **SMALLER** than pivot.

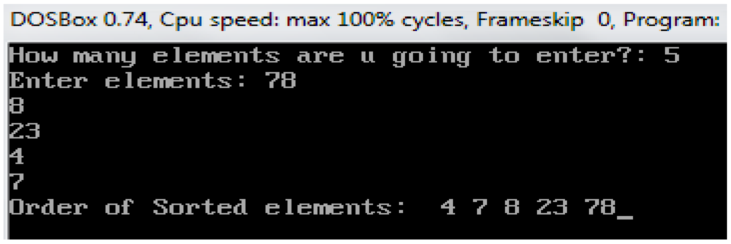




Example :

```
#include<stdio.h>
void quicksort(int number[25],int first,int last)
{
  int i, j, pivot, temp;

  if(first<last)
  {
    pivot=first;
    i=first;
    j=last;

    while(i<j)
    {
      while(number[i]<=number[pivot]&&i<last)
        i++;
      while(number[j]>number[pivot])
        j--;
      if(i<j)
      {
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }

    temp=number[pivot];
    number[pivot]=number[j];
```

```
      number[j]=temp;
      quicksort(number,first,j-1);
      quicksort(number,j+1,last);

   }
}

int main()
{
   int i,n, number[25];

   printf("How many elements are u going to enter?: ");
   scanf("%d",&n);

   printf("Enter elements: ");
   for(i=0;i<n;i++)
      scanf("%d",&number[i]);

   quicksort(number,0,n-1);

   printf("Order of Sorted elements: ");
   for(i=0;i<n;i++)
      printf(" %d",number[i]);
  return 0;
}
```



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program:

```
How many elements are u going to enter?: 5
Enter elements: 78
8
23
4
7
Order of Sorted elements:  4 7 8 23 78_
```

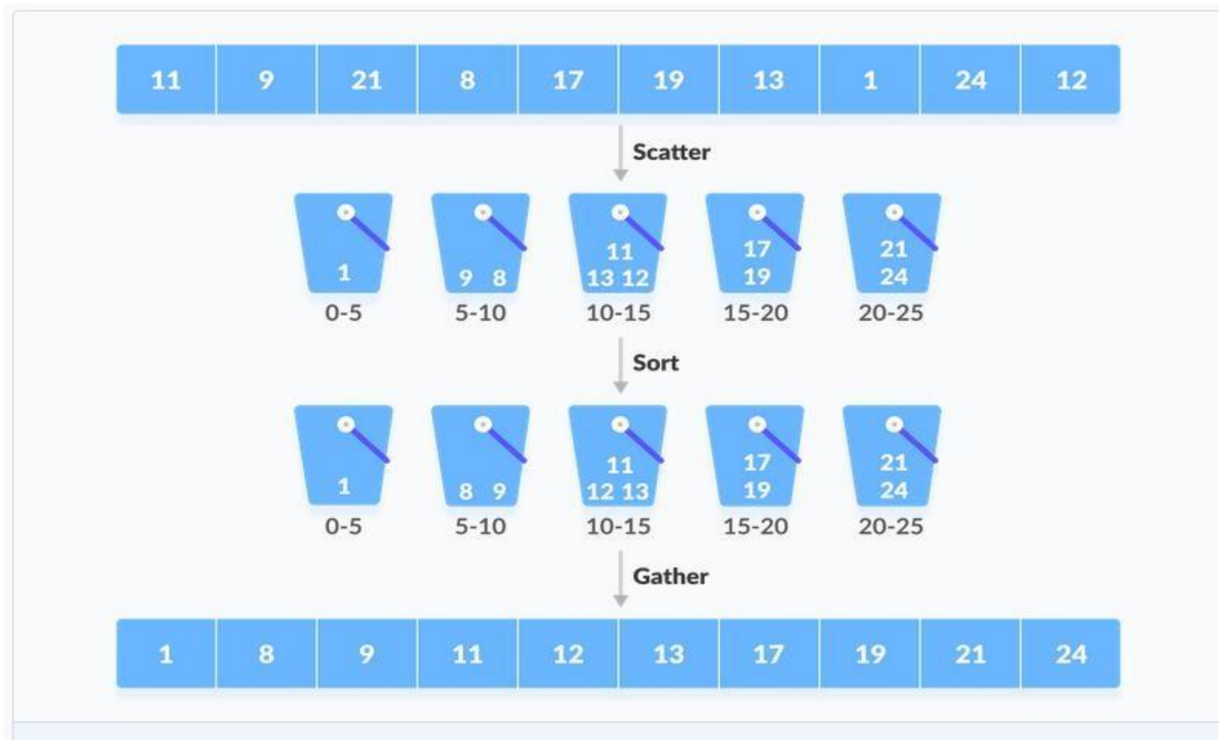
7. Bucket sort

Bucket Sort is a sorting technique that sorts the elements by first dividing the elements into several groups called **buckets**. The elements inside each **bucket** are sorted using any of the suitable sorting algorithms or recursively calling the same algorithm.

Several buckets are created. **Each bucket is filled with a specific range of elements**.

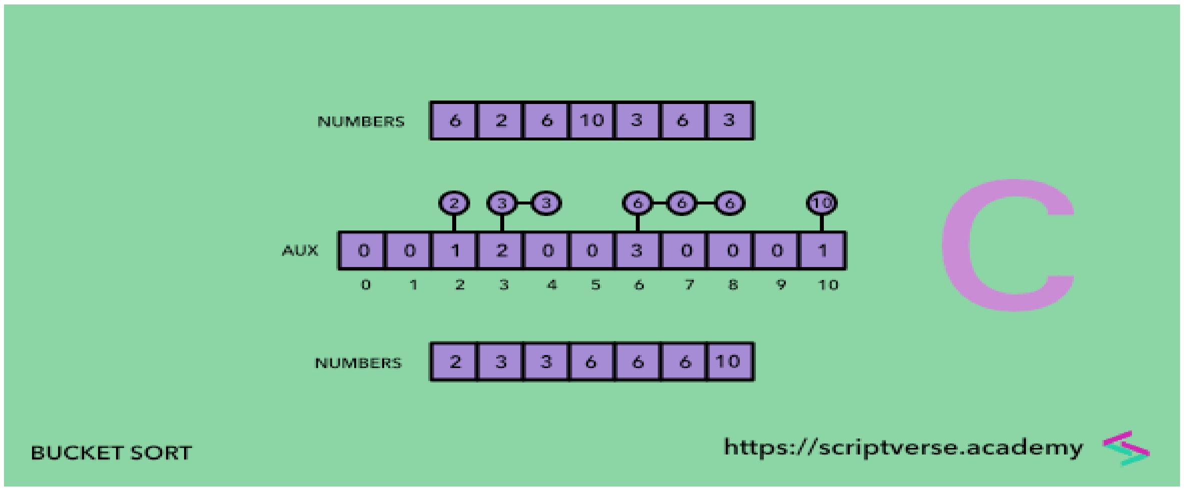
The elements inside the bucket are sorted using any other algorithm.

Finally, the elements of the bucket are gathered to get the sorted array.



Working of Bucket Sort

Bucket sort (or bin sort) works by distributing the elements into a number of buckets, and each bucket is then sorted individually. Bucket sort is a distribution sort, and is a cousin of radix sort. Bucket sort is not a comparison sort like bubble sort, insertion sort or selection sort.



The algorithm is as follows:

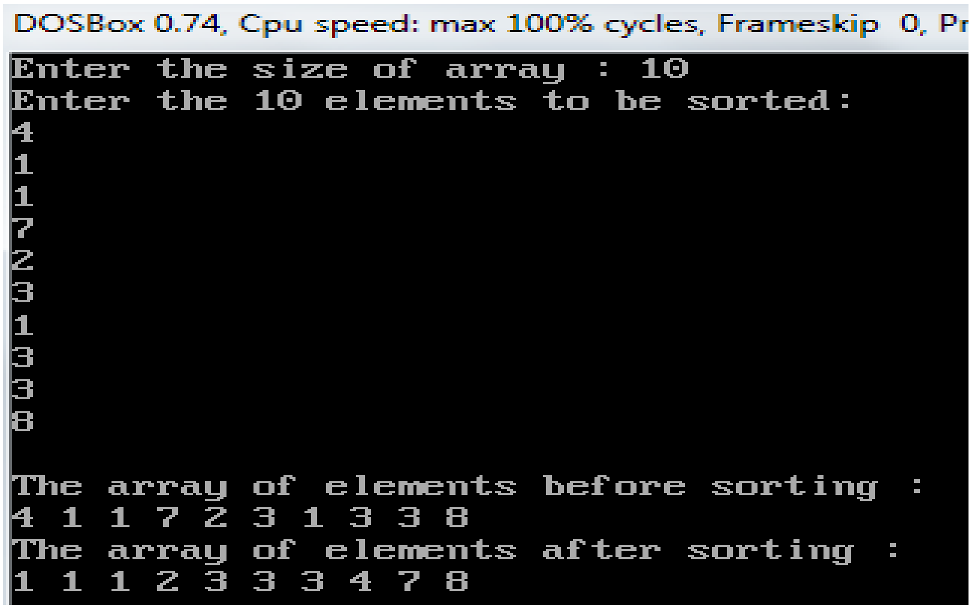
1. Set up buckets, initially empty.
2. Scatter: place each element into an appropriate bucket.
3. Sort each non-empty bucket.
4. Gather: Gather elements from buckets and put back to the original array.

Example

```
#include <stdio.h>
void Bucket_Sort(int array[], int n)
{

   int i, j;
   int count[100];
   for (i = 0; i < n; i++)
```

```
        count[i] = 0;

    for (i = 0; i < n; i++)
        (count[array[i]])++;

    for (i = 0, j = 0; i < n; i++)
        for(; count[i] > 0; (count[i])--)
            array[j++] = i;

}
int main()
{
    int array[100], i, num;
    printf("Enter the size of array : ");
    scanf("%d", &num);

    printf("Enter the %d elements to be sorted:\n",num);
    for (i = 0; i < num; i++)
        scanf("%d", &array[i]);

    printf("\nThe array of elements before sorting : \n");
    for (i = 0; i < num; i++)
        printf("%d ", array[i]);

    printf("\nThe array of elements after sorting : \n");
    Bucket_Sort(array, num);
    for (i = 0; i < num; i++)
        printf("%d ", array[i]);
    printf("\n");
    return 0;

}
```

Output



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Pr

```
Enter the size of array : 10
Enter the 10 elements to be sorted:
4
1
1
7
2
3
1
3
3
8

The array of elements before sorting :
4 1 1 7 2 3 1 3 3 8
The array of elements after sorting :
1 1 1 2 3 3 3 4 7 8
```

Searching Techniques

Searching is the process of **finding some particular element in the list.**

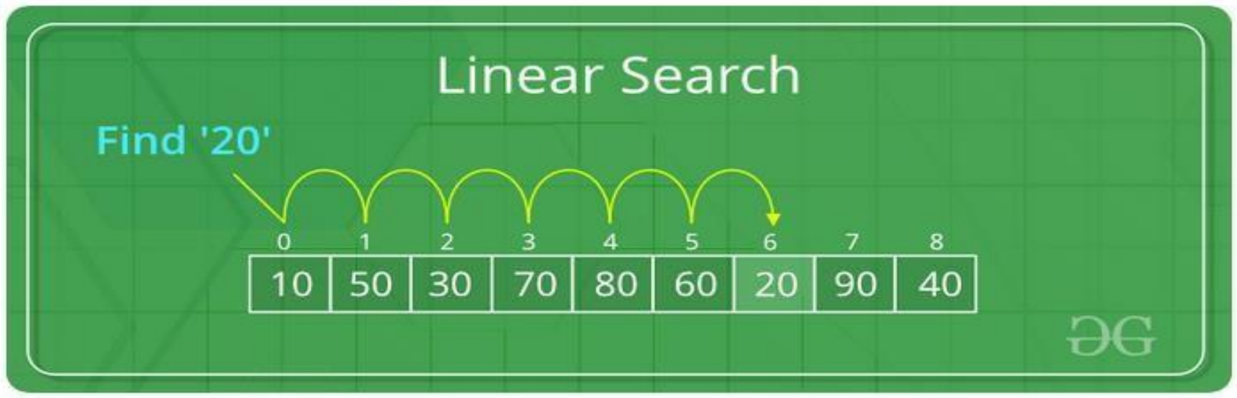
The process of identifying or finding a particular record is called Searching.

If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

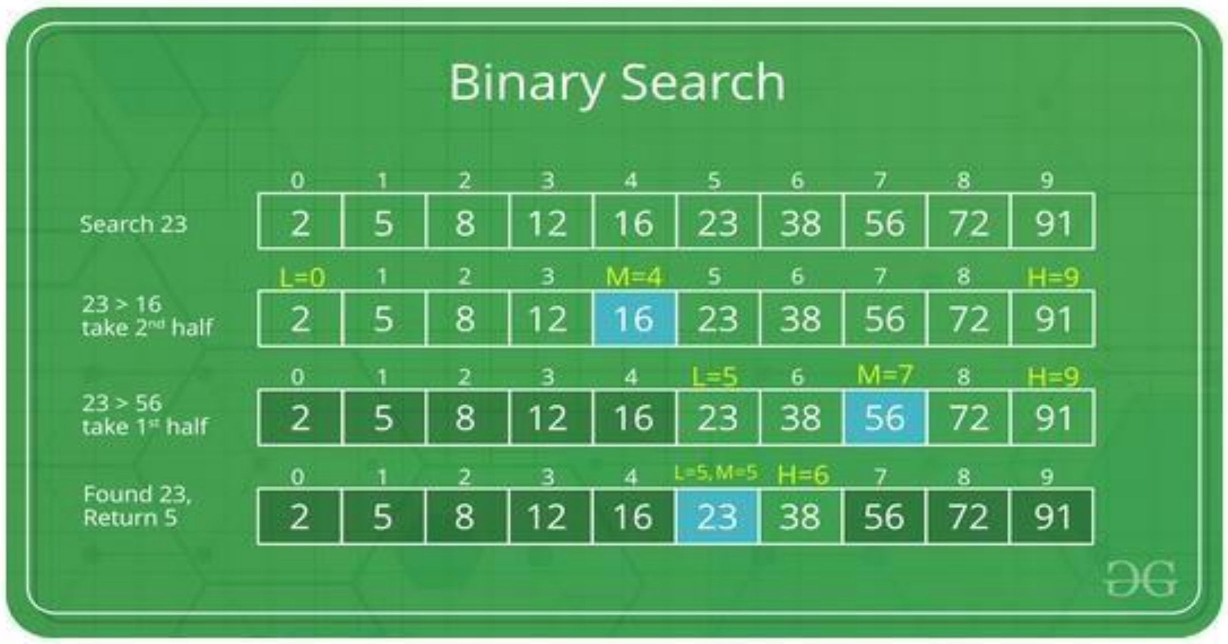
1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: **Linear Search.**

Linear Search to find the element "20" in a given list of numbers



2. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: **Binary Search.**

Binary Search to find the element "23" in a given list of numbers



1. Sequential Searching

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

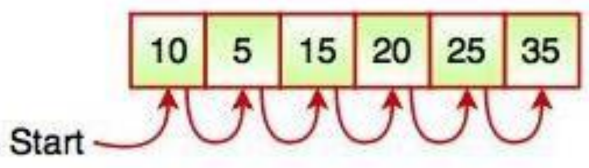
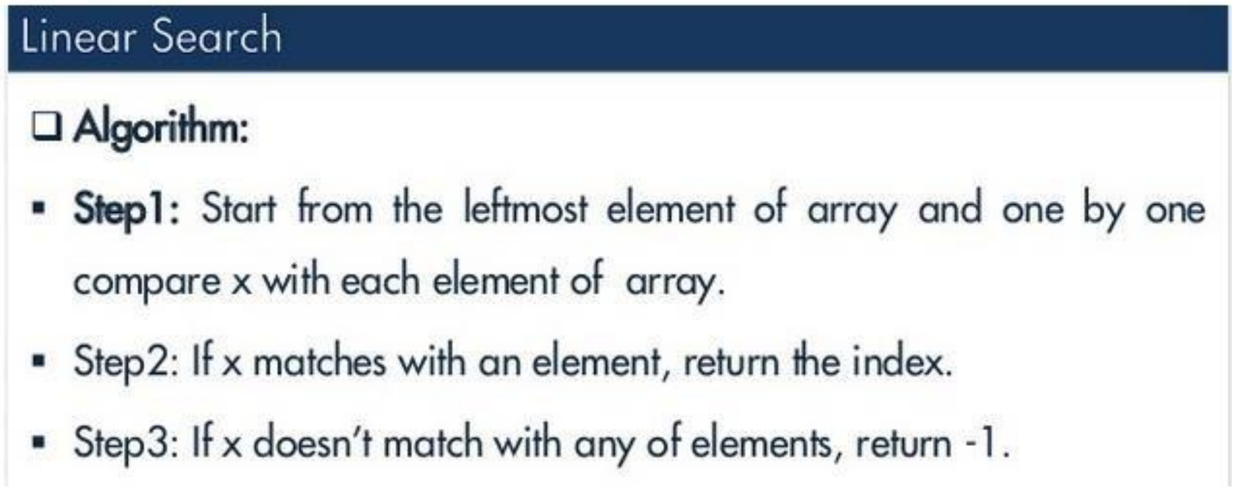


Fig. Sequential Search

- The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order.
- It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.



Linear Search

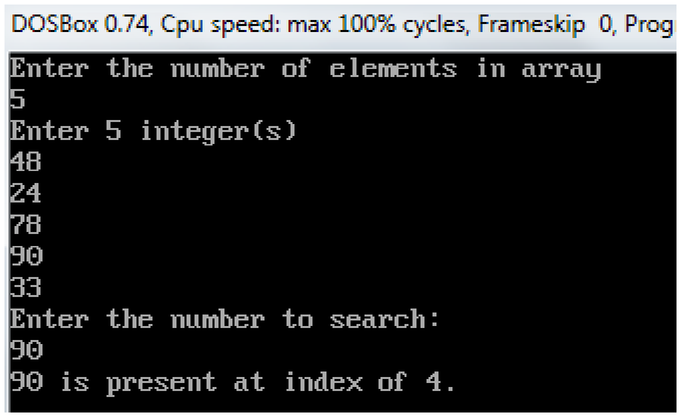
❑ **Algorithm:**

- **Step1:** Start from the leftmost element of array and one by one compare x with each element of array.
- Step2: If x matches with an element, return the index.
- Step3: If x doesn't match with any of elements, return -1.

<u>Example :</u>

```
#include <stdio.h>
int main()
{
  int arr[50], search, cnt, num;
  clrscr();
  printf("Enter the number of elements in array\n");
  scanf("%d",&num);

  printf("Enter %d integer(s)\n", num);

  for (cnt = 1; cnt <= num; cnt++)
  scanf("%d", &arr[cnt]);

  printf("Enter the number to search:\n");
  scanf("%d", &search);

  for (cnt = 0; cnt < num; cnt++)
  {
    if (arr[cnt] == search)     /* if required element found */
    {
      printf("%d is present at index of %d.\n", search, cnt+1);
      break;
    }
  }
  if (cnt >= num)
    printf("%d is not present in array.\n", search);

  return 0;
}
```

Output :

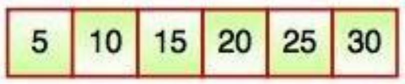

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Prog

```
Enter the number of elements in array
5
Enter 5 integer(s)
48
24
78
90
33
Enter the number to search:
90
90 is present at index of 4.
```

2. Binary Searching

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of O(log n).
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.

- It is useful when there are large number of elements in an array.



- The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

For example, if searching an element 25 in the 7-element array, following figure shows how binary search works:

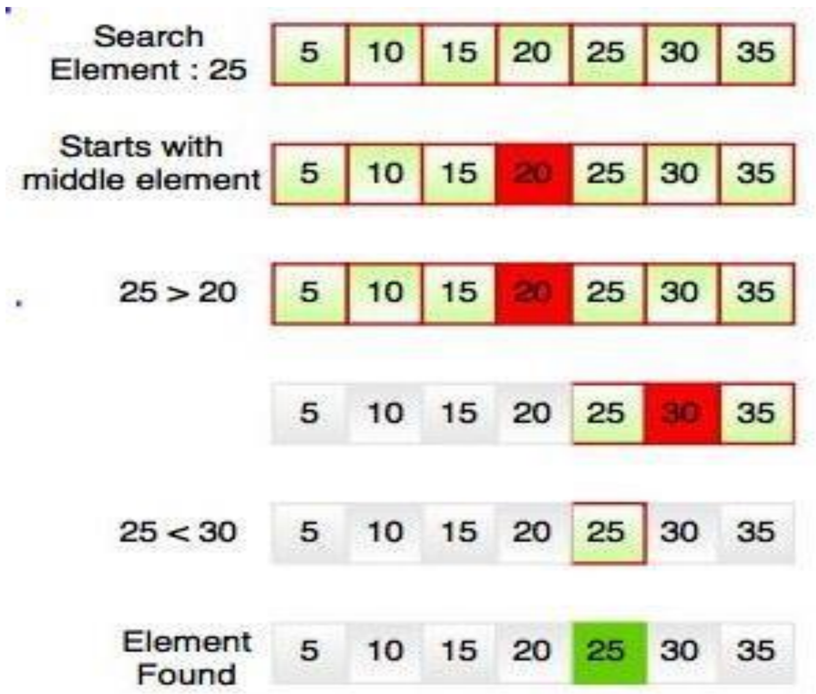
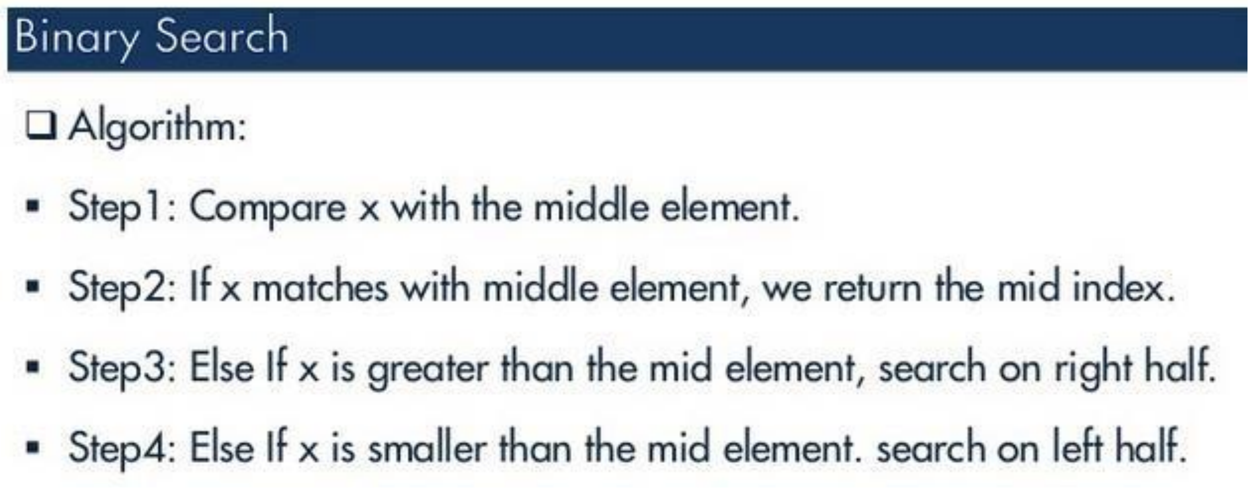


Fig. Working Structure of Binary Search

- Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.



Binary Search

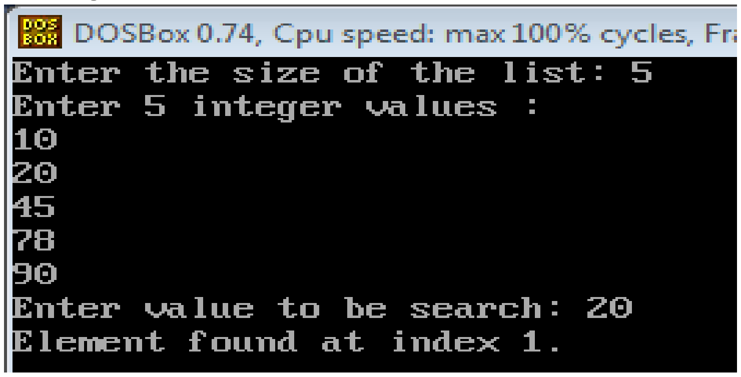
❑ Algorithm:

- Step1: Compare x with the middle element.
- Step2: If x matches with middle element, we return the mid index.
- Step3: Else If x is greater than the mid element, search on right half.
- Step4: Else If x is smaller than the mid element. search on left half.

Example :

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

void main()
{
  int f, l, m, size, i, sElement, list[50]; //int f, l ,m : First, Last, Middle
  clrscr();

  printf("Enter the size of the list: ");
  scanf("%d",&size);

  printf("Enter %d integer values : \n", size);

  for (i = 0; i < size; i++)
    scanf("%d",&list[i]);

  printf("Enter value to be search: ");
  scanf("%d", &sElement);

  f = 0;
  l = size - 1;
  m = (f+l)/2;

  while (f <= l)
  {
    if (list[m] < sElement)
      f = m + 1;
    else if (list[m] == sElement)
     {
      printf("Element found at index %d.\n",m);
      break;
     }
    else
      l = m - 1;
      m = (f + l)/2;
   }
  if (f > l)
    printf("Element Not found in the list.");
  getch();
}
```

Output :



```
Enter the size of the list: 5
Enter 5 integer values :
10
20
45
78
90
Enter value to be search: 20
Element found at index 1.
```

3. Index Searching

Indexing is **a data structure technique that helps to speed up data retrieval**.

Index search is special search. This search method is used to search a record in a file.

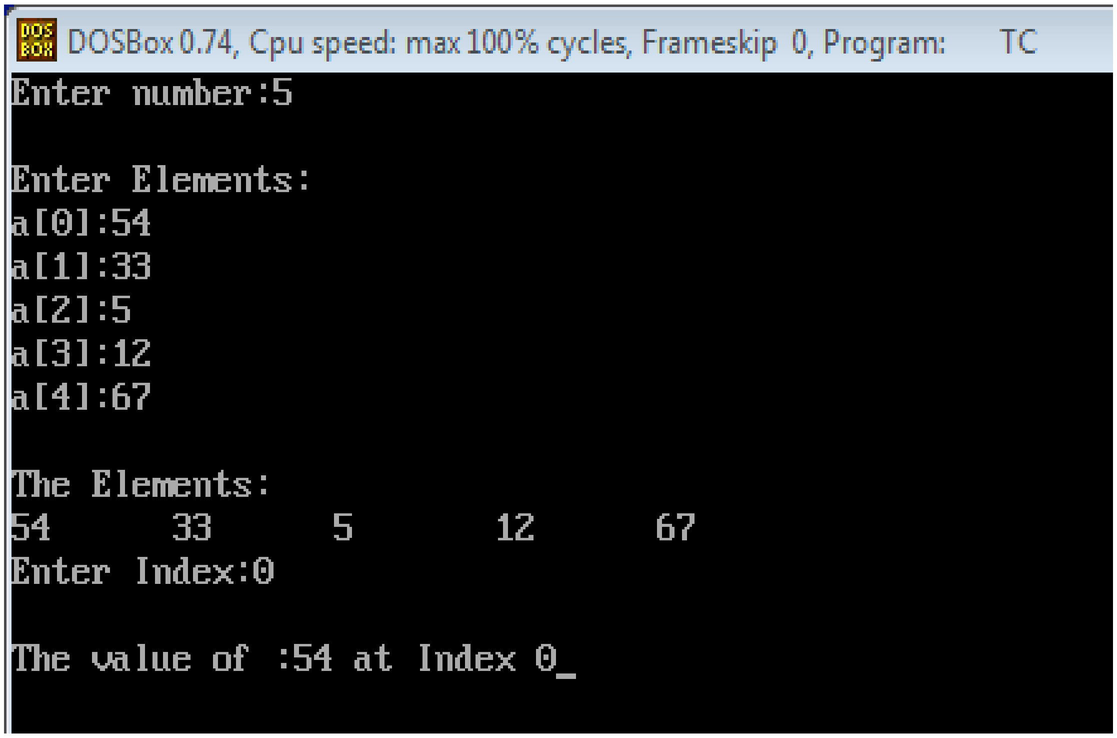
Searching a record refers to the searching of location loc in memory where the file is stored.

Indexed search searches the record with a given key value relative to a primary key field.

Example

```
#include<stdio.h>
#include<conio.h>
int Index_Search(int a[],int n,int s)
{
        int i;
        for(i=0;i<n;i++)
        {
                if(i==s)
                        break;
        }
        return a[s];
}
void main()
{
        int a[100],n,i,s,val;
        clrscr();
        printf("Enter number:");
        scanf("%d",&n);
        printf("\nEnter Elements:\n");
        for(i=0;i<n;i++)
        {
                printf("a[%d]:",i);
                scanf("%d",&a[i]);
        }
        printf("\nThe Elements:\n");
        for(i=0;i<n;i++)
        {
                printf("%d\t",a[i]);
        }
        printf("\nEnter Index:");
        scanf("%d",&s);
        val=Index_Search(a,n,s);
        printf("\nThe value of :%d at Index %d",val,s);
        getch();
}
```

Output



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Enter number:5

Enter Elements:
a[0]:54
a[1]:33
a[2]:5
a[3]:12
a[4]:67

The Elements:
54      33      5       12      67
Enter Index:0

The value of :54 at Index 0_
```